Языки программирования. Концепции и принципы

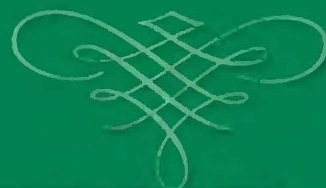
Кауфман В. Ш.

Рассмотрены основные концепции и принципы, воплощенные в современных и перспективных языках программирования (Фортран, Паскаль, ПЛ/1, Алгол-68, Симула-67, Смолток, Рефал, Ада, Модула-2, Оберон, Оккам-2, Турбо Паскаль 5.5 и др.), представлены разные стили программирования (операторный, ситуационный, функциональный, реляционный, параллельный, объектно-ориентированный), освещены тенденции и перспективы развития языков и стилей программирования.

Впервые удалось разработать и представить в одной книге цельную систему концепций и принципов, создающих достаточно четкие ориентиры в области языков программирования. На основе этой системы сформулированы оригинальные положения, указывающие перспективы развития языков программирования (модули исключительных ситуаций, модули управления представлением, входовые типы и др.).

Новые подходы применены при изложении известных фактов (пошаговая модификация нормальных алгоритмов Маркова сначала до Рефала, а затем до реляционных языков, систематическое сопоставление концепции параллелизма в Аде и Оккаме-2, концепций создания Ады, Модулы-2 и Оберона, развитие концепции наследуемости от модульности до объектной ориентации и др.).

Для научных работников, будет полезной программистам, а также преподавателям и студентам, серьезно интересующимся языками программирования.



Internet-магазин: www.alians-kniga.ru

Книга-почтой:
Россия, 123242, Москва, а/я 20
e-mail: orders@alians-kniga.ru

Оптовая продажа: "Альянс-книга"
(495)258-9194, 258-9195
e-mail: books@alians-kniga.ru

978-5-94074-622-5
9 785940 746225

Классика программирования

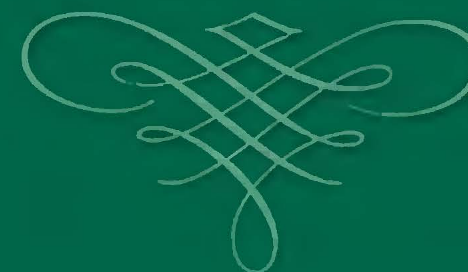
Кауфман В. Ш.

Языки программирования

Классика программирования

Кауфман В. Ш.

Языки программирования. Концепции и принципы



В. Ш. Кауфман

Языки программирования

Концепции и принципы

Москва, 2011

УДК 519.682.1
ББК 004.438
К30

Рецензент О. Н. Перминов

К30 Кауфман В. Ш.
Языки программирования. Концепции и принципы. – М.: ДМК Пресс, 2010. – 464 с.: ил.

ISBN 978-5-94074-622-5

Рассмотрены фундаментальные концепции и принципы, воплощенные в современных и перспективных языках программирования. Представлены разные стили программирования (операционный, ситуационный, функциональный, реляционный, параллельный, объектно-ориентированный). Базовые концепции и принципы рассмотрены с пяти различных позиций (технологической, авторской, математической, семиотической и реализаторской) и проиллюстрированы примерами из таких языков, как Паскаль, Симула-67, Смолток, Рефал, Ада, Модула-2, Оберон, Оккам-2, Турбо Паскаль, С++ и др.

Сложность выделена как основополагающая проблема программирования, а абстракция-конкретизация и прогнозирование-контроль – как основные ортогональные методы борьбы со сложностью. На этой общей базе в книге впервые представлена цельная система концепций и принципов, создающая четкие ориентиры в области языков программирования. На основе этой системы сформулированы оригинальные положения, указывающие перспективы развития в этой области (модули исключительных ситуаций, модули управления представлением, входовыетипы и др.). Многие из них в последние годы стали реальностью.

Новые подходы применены при изложении известных фактов (пошаговая модификация нормальных алгоритмов Маркова сначала до Рефала, а затем до реляционных языков, сопоставление принципов «сундука» и «чемоданчика» при создании Ады, Модулы-2 и Оберона, развитие концепции наследуемости от модульности до объектной ориентации, систематическое сопоставление концепции параллелизма в Аде и Оккаме-2, и др.).

Для всех, серьезно интересующихся программированием, в том числе научных работников, программистов, преподавателей и студентов.

Ил. 5 Библиогр. 64 назв.

УДК 519.682.1
ББК 004.438

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-94074-622-5

© Кауфман В. Ш., 2010
© Оформление, издание, ДМК Пресс, 2011

Краткое содержание

Предисловие ко второму изданию ... 14

Предисловие ... 15

ЧАСТЬ I. СОВРЕМЕННОЕ СОСТОЯНИЕ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ... 19

ГЛАВА 1. КОНЦЕПТУАЛЬНАЯ СХЕМА ЯЗЫКА ПРОГРАММИРОВАНИЯ ... 21

ГЛАВА 2. ПРИМЕР СОВРЕМЕННОГО БАЗОВОГО ЯП (МОДЕЛЬ А) ... 43

ГЛАВА 3. ВАЖНЕЙШИЕ АБСТРАКЦИИ: ДАННЫЕ, ОПЕРАЦИИ, СВЯЗЫВАНИЕ ... 69

ГЛАВА 4. ДАННЫЕ И ТИПЫ ... 85

ГЛАВА 5. РАЗДЕЛЬНАЯ КОМПИЛЯЦИЯ ... 145

ГЛАВА 6. АСИНХРОННЫЕ ПРОЦЕССЫ ... 151

ГЛАВА 7. НОТАЦИЯ ... 175

ГЛАВА 8. ИСКЛЮЧЕНИЯ ... 183

ГЛАВА 9. БИБЛИОТЕКА ... 201

ГЛАВА 10. ИМЕНОВАНИЕ И ВИДИМОСТЬ (НА ПРИМЕРЕ АДЫ) ... 205

ГЛАВА 11. ОБМЕН С ВНЕШНЕЙ СРЕДОЙ ... 219

ГЛАВА 12. ДВА АЛЬТЕРНАТИВНЫХ ПРИНЦИПА СОЗДАНИЯ ЯП ... 237

ЧАСТЬ II. ПЕРСПЕКТИВЫ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ... 267

ГЛАВА 1. ПЕРСПЕКТИВНЫЕ МОДЕЛИ ЯЗЫКА 269

ГЛАВА 2. ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ (МОДЕЛЬ Б) 289

ГЛАВА 3. ДОКАЗАТЕЛЬНОЕ ПРОГРАММИРОВАНИЕ (МОДЕЛЬ Д) 315

ГЛАВА 4. РЕЛЯЦИОННОЕ ПРОГРАММИРОВАНИЕ (МОДЕЛЬ Р) 339

ГЛАВА 5. ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В ОККАМЕ-2 (МОДЕЛЬ О) 353

ГЛАВА 6. НАСЛЕДУЕМОСТЬ (К ИДЕАЛУ РАЗВИТИЯ И ЗАЩИТЫ В ЯП) 391

ГЛАВА 7. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ 415

ГЛАВА 8. ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ 439

Заключение 459

Список литературы 460

Полезная литература, на которую прямых ссылок в тексте нет 463

Содержание

Предисловие ко второму изданию 14

Предисловие 15

Часть I. СОВРЕМЕННОЕ СОСТОЯНИЕ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ 19

Глава 1. Концептуальная схема языка программирования 21

1.1. Что такое язык программирования 22
1.2. Метауровень 22
1.3. Модель передачи сообщения 23
1.4. Классификация недоразумений 23
1.5. Отступление об абстракции-конкретизации. Понятие модели 25
1.6. Синтактика, семантика, прагматика 26
1.7. Зачем могут понадобиться знания о ЯП 27
1.8. Принцип моделирования ЯП 29
1.9. Пять основных позиций рассмотрения ЯП 29
1.10. Что такое производство программных услуг 30
1.11. Производство программных услуг – основная цель программирования 32
1.12. Сложность как основная проблема программирования 33
1.13. Источники сложности 34
1.14. Два основных средства борьбы со сложностью. Основной критерий качества ЯП 36
1.15. Язык программирования как знаковая система 37
1.16. Разновидности программирования 38
1.17. Понятие о базовом языке 39
1.18. Концептуальная схема рассмотрения ЯП 40

Глава 2. Пример современного базового ЯП (модель А) 43

2.1. Общее представление о ЯП Ада 44
2.2. Пример простой программы на Аде 46
2.3. Обзор языка Ада 47
  2.3.1. Модули 48
  2.3.2. Объявления и операторы 49
  2.3.3. Типы данных 50
2.4. Пошаговая детализация средствами Ады 52
2.5. Замечания о конструктах 57
2.6. Как пользоваться пакетом управление_сетью 59

2.7. Принцип раздельного определения, реализации и использования услуг (принцип РОРИУС) 66
2.8. Принцип защиты абстракций 67

Глава 3. Важнейшие абстракции: данные, операции, связывание 69

3.1. Принцип единства и относительности трех абстракций 70
3.2. Связывание 71
3.3. От связывания к пакету 72
3.4. Связывание и специализация 74
3.4.1. Связывание и теория трансляции 75
3.5. Принцип цельности 79
3.5.1. Принцип цельности и нормальные алгоритмы 81
3.5.2. Принцип цельности и Ада. Критерий цельности 82

Глава 4. Данные и типы 85

4.1. Классификация данных 86
4.2. Типы данных 88
4.2.1. Динамические, статические и относительно статические ЯП 88
4.2.2. Система типов как знаковая система 90
4.2.3. Строгая типизация и уникальность типа 93
4.2.4. Критичные проблемы, связанные с типами 93
4.2.5. Критичные потребности и критичные языковые проблемы 94
4.2.6. Проблема полиморфизма 94
4.2.7. Янус-проблема 96
4.2.8. Критерий содержательной полноты ЯП. Неформальные теоремы 98
4.3. Регламентированный доступ и типы данных 98
4.3.1. Задача моделирования многих сетей 99
4.3.2. Приватные типы данных 101
4.3.3. Строго регламентированный доступ. Ограниченные приватные типы 103
4.3.4. Инкапсуляция 105
4.4. Характеристики, связанные с типом. Класс значений, базовый набор операций 106
4.5. Воплощение концепции уникальности типа. Определение и использование типа в Аде (начало) 107
4.5.1. Объявление типа. Конструктор типа. Определяющий пакет 107
4.6. Конкретные категории типов 108
4.6.1. Перечисляемые типы. «Морская задача» 108
4.6.2. Дискретные типы 116
4.6.3. Ограничения и подтипы 118
4.6.4. Квазистатический контроль 120
4.6.5. Подтипы 122

4.6.6. Принцип целостности объектов 123
4.6.7. Объявление подтипа 125
4.6.8. Подтипы и производные типы. Преобразования типа 125
4.6.9. Ссылочные типы (динамические объекты) 127
4.7. Типы как объекты высшего порядка. Атрибутные функции 129
4.7.1. Статическая определимость типа 129
4.7.2. Почему высшего порядка? 129
4.7.3. Действия с типами 129
4.8. Родовые (настраиваемые) сегменты 131
4.9. Числовые типы (модель числовых расчетов) 133
4.9.1. Суть проблемы 133
4.9.2. Назначение модели расчетов 134
4.9.3. Классификация числовых данных 134
4.9.4. Зачем объявлять диапазон и точность 135
4.9.5. Единая модель числовых расчетов 135
4.9.6. Допустимые числа 136
4.10. Управление операциями 137
4.11. Управление представлением 138
4.12. Классификация данных и система типов Ады 141
4.13. Предварительный итог по модели А 143

Глава 5. Раздельная компиляция 145

5.1. Понятие модуля 146
5.2. Виды трансляций 146
5.3. Раздельная трансляция 146
5.4. Связывание трансляционных модулей 147
5.4.1. Модули в Аде 147
5.5. Принцип защиты авторского права 148

Глава 6. Асинхронные процессы 151

6.1. Основные проблемы 152
6.2. Семафоры Дейкстры 155
6.3. Сигналы 157
6.4. Концепция внешней дисциплины 159
6.5. Концепция внутренней дисциплины: мониторы 160
6.6. Рандеву 164
6.7. Проблемы рандеву 165
6.8. Асимметричное рандеву 166
6.9. Управление асимметричным рандеву (семантика вспомогательных конструктов) 167
6.10. Реализация семафоров, сигналов и мониторов посредством асимметричного рандеву 169
6.11. Управление асинхронными процессами в Аде 172

Глава 7. Нотация ... 175

7.1. Проблема знака в ЯП ... 176
7.2. Определяющая потребность ... 176
7.3. Основная абстракция ... 177
7.4. Проблема конкретизации эталонного текста ... 177
7.5. Стандартизация алфавита ... 178
7.6. Основное подмножество алфавита ... 179
7.7. Алфавит языка Ада ... 179
7.8. Лексемы ... 180
7.9. Лексемы в Аде ... 181

Глава 8. Исключения ... 183

8.1. Основная абстракция ... 184
8.2. Определяющие требования ... 185
8.3. Аппарат исключений в ЯП ... 187
 8.3.1. Определение исключений ... 187
 8.3.2. Распространение исключений. Принцип динамической ловушки ... 189
 8.3.3. Реакция на исключение – принципы пластыря и катапульты ... 191
 8.3.4. Ловушка исключений ... 193
8.4. Дополнительные особенности обработки исключений ... 194

Глава 9. Библиотека ... 201

9.1. Структура библиотеки ... 202
9.2. Компилируемый (трансляционный) модуль ... 202
9.3. Порядок компиляции и перекомпиляции (создания и модификации программной библиотеки) ... 203
9.4. Резюме: логическая и физическая структуры программы ... 204

Глава 10. Именование и видимость (на примере Ады) ... 205

10.1. Имя как специфический знак ... 206
10.2. Имя и идентификатор ... 206
10.3. Проблема видимости ... 206
10.4. Аспекты именования ... 207
10.5. Основная потребность и определяющие требования ... 207
10.6. Конструкты и требования, связанные с именованием ... 208
10.7. Схема идентификации ... 210
 10.7.1. Виды объявлений в Аде ... 210
 10.7.2. Области локализации и «пространство имен» Ада-программы ... 213
 10.7.3. Область непосредственной видимости ... 215
 10.7.4. Идентификация простого имени ... 216
 10.7.5. Идентификация составного имени ... 216
10.8. Недостатки именования в Аде ... 216

Глава 11. Обмен с внешней средой 219

11.1. Специфика обмена 220
11.2. Назначение и структура аппарата обмена 223
11.2.1. Файловая модель 224
11.3. Файловая модель обмена в Аде 224
11.3.1. Последовательный обмен 225
11.3.2. Комментарий 226
11.3.3. Пример обмена. Программа диалога 229
11.3.4. Отступление о видимости и родовых пакетах 231
11.4. Программирование специальных устройств 233

Глава 12. Два альтернативных принципа создания ЯП ... 237

12.1. Принцип сундука 238
12.2. Закон распространения сложности ЯП 238
12.3. Принцип чемоданчика 239
12.4. Обзор языка Модула-2 239
12.4.1. Характеристика Модулы-2 в координатах фон-неймановского языкового пространства (технологическая позиция) 240
12.5. Пример М-программы 241
12.5.1. Управление сетями на Модуле-2 242
12.5.2. Определяющий модуль 242
12.5.3. Использующий модуль 244
12.5.4. Реализующий модуль 245
12.6. Языковая ниша 248
12.7. Принцип чемоданчика в проектных решениях ЯП Модула-2 249
12.7.1. Видимость 249
12.7.2. Инкапсуляция 251
12.7.3. Обмен 251
12.8. Принцип чайника 257
12.9. ЯП Оберон 258
12.9.1. От Модулы-2 к Оберону 259
12.9.2. Управление сетями на Обероне 261

Часть II. ПЕРСПЕКТИВЫ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ 267

Глава 1. Перспективные модели языка 269

1.1. Введение 270
1.2. Операционное программирование – модель фон Неймана (модель Н) 271
1.3. Ситуационное программирование – модель Маркова-Турчина (модель МТ) 273
1.3.1. Перевод в польскую инверсную запись (ПОЛИЗ) 274
1.3.2. Модификации модели Маркова (введение в Рефал) 275

1.3.3. Исполнитель (МТ-машина) 280
1.3.4. Программирование в модели МТ 280
1.3.5. Основное семантическое соотношение в модели МТ 281
1.3.6. Пример вычисления в модели МТ 283
1.3.7. Аппликативное программирование 285
1.3.8. Структуризация поля определений. МТ-функции 286

Глава 2. Функциональное программирование (модель Б) 289

2.1. Функциональное программирование в модели МТ 290
2.1.1. Модель МТ с точки зрения концептуальной схемы 290
2.1.2. Модель МТ и Лисп 291
2.1.3. Критерий концептуальной ясности и функции высших порядков 291
2.1.4. Зачем нужны функции высших порядков 292
2.1.5. Примеры структурирующих форм 294
2.1.6. Пример программы в стиле Бэкуса 297
2.2. Функциональное программирование в стиле Бэкуса (модель Б) 299
2.2.1. Модель Бэкуса с точки зрения концептуальной схемы 299
2.2.2. Объекты 300
2.2.3. Аппликация 300
2.2.4. Функции 301
2.2.5. Условные выражения Маккарти 301
2.2.6. Примеры примитивных функций 302
2.2.7. Примеры форм, частично известных по работе в модели МТ 304
2.2.8. Определения 306
2.2.9. Программа вычисления факториала 306
2.2.10. Программа перемножения матриц 308

Глава 3. Доказательное программирование (модель Д) 315

3.1. Зачем оно нужно 316
3.2. Доказательное программирование методом Бэкуса 316
3.2.1. Алгебра программ в модели Б 317
3.2.2. Эквивалентность двух программ перемножения матриц 318
3.3. Доказательное программирование методом Хоара 321
3.3.1. Модель Д 322
3.3.2. Дедуктивная семантика 324
3.3.3. Компоненты исчисления Хоара 326
3.3.4. Правила преодоления конструктов языка Д 328
3.3.5. Применение дедуктивной семантики 334

Глава 4. Реляционное программирование (модель Р) 339

4.1. Предпосылки 340
4.2. Ключевая идея 341

4.3. Пример ... 341
4.3.1. База данных ... 342
4.3.2. База знаний ... 342
4.3.3. Пополнение базы данных (вывод фактов) ... 343
4.3.4. Решение задач ... 344
4.3.5. Управление посредством целей ... 344
4.4. О предопределенных отношениях ... 347
4.5. Связь с моделями МТ и Б ... 348
4.5.1. Путь от модели Б ... 348
4.5.2. Путь от модели МТ ... 350

Глава 5. Параллельное программирование в Оккаме-2 (модель О) ... 353

5.1. Принципы параллелизма в Оккаме ... 354
5.2. Первые примеры применения каналов ... 355
5.3. Сортировка конвейером фильтров ... 356
5.4. Параллельное преобразование координат (умножение вектора на матрицу) ... 357
5.4.1. Структура коллектива процессов ... 358
5.4.2. Коммутация каналов ... 361
5.5. Монитор Хансена-Хоара на Оккаме-2 ... 362
5.6. Сортировка деревом исполнителей ... 363
5.7. Завершение работы коллектива процессов ... 366
5.8. Сопоставление концепций параллелизма в Оккаме и в Аде ... 368
5.8.1. Концепция параллелизма в Аде ... 369
5.8.2. Параллельное преобразование координат в Аде ... 371
5.9. Перечень неформальных теорем о параллелизме в Аде и Оккаме ... 377
5.10. Единая модель временных расчетов ... 378
5.11. Моделирование каналов средствами Ады ... 378
5.12. Отступление о задачных и подпрограммных (процедурных) типах ... 381
5.12.1. Входовые типы – фрагмент авторской позиции ... 381
5.12.2. Обоснование входовых типов ... 384
5.12.3. Родовые подпрограммные параметры ... 386
5.12.4. Почему же в Аде нет подпрограммных типов? ... 387
5.12.5. Заключительные замечания ... 387

Глава 6. Наследуемость (к идеалу развития и защиты в ЯП) ... 391

6.1. Определяющая потребность ... 392
6.2. Идеал развиваемости ... 392
6.3. Критичность развиваемости ... 393
6.4. Аспекты развиваемости ... 393
6.5. Идеал наследуемости (основные требования) ... 395
6.6. Проблема дополнительных атрибутов ... 395

6.7. Развитая наследуемость 398
6.8. Аспект данных 398
6.9. Аспект операций 401
6.10. Концепция наследования в ЯП (краткий обзор) 407
6.10.1. Основные понятия и неформальные аксиомы наследования 407
6.11. Преимущества развитой наследуемости 409
6.12. Наследуемость и гомоморфизм (фрагмент математической позиции) 410

Глава 7. Объектно-ориентированное программирование 415

7.1. Определяющая потребность 416
7.2. Ключевые идеи объектно-ориентированного программирования 417
7.3. Пример: обогащение сетей на Турбо Паскале 5.5 418
7.4. Виртуальные операции 423
7.5. Критерий Дейкстры 431
7.6. Объекты и классы в ЯП Симула-67 432
7.7. Перспективы, открываемые объектной ориентацией средств программирования 434
7.8. Свойства объектной ориентации 437
7.9. Критерий фундаментальности языковых концепций 438

Глава 8. Заключительные замечания 439

8.1. Реализаторская позиция 440
8.1.1. Компоненты реализации 440
8.1.2. Компиляторы 443
8.1.3. Основная функция компилятора 444
8.1.4. Три принципа создания компиляторов 445
8.2. Классификация языков программирования 448
8.2.1. Традиционная классификация 448
8.2.2. Недостатки традиционной классификации 450
8.2.3. Принцип инерции программной среды 450
8.2.4. Заповеди программиста 451
8.3. Тенденции развития ЯП 451
8.3.1. Перспективные абстракции 451
8.3.2. Абстракция от программы (в концептуальном и реляционном программировании) 455
8.3.3. Социальный аспект ЯП 457
8.3.4. Стандартизация ЯП 457

Заключение 459

Список литературы 460

Полезная литература, на которую прямых ссылок в тексте нет 463

Моим дорогим родителям
Александре Фоминичне Каревой
Шахно Мордуховичу Кауфману
Эдит Яковлевне Кауфман

Предисловие ко второму изданию

К немалому удивлению автора, это книга оказалась востребованной и через 17 лет после своего официального выхода в свет (насколько известно автору, ее используют в МГУ, МАИ и других российских университетах). Это тем более удивительно, что основной ее материал подготовлен значительно раньше, примерно в 1985 году.

В Сети также циркулируют (и даже продаются, якобы с разрешения автора ☺) относительно ранние варианты лекций по курсу «Языки программирования», читанных автором в те же годы на факультете ВМиК МГУ.

Многие уважаемые члены программистского сообщества посчитали нужным поддержать уверенность автора в ценности изложенного в книге материала.

Владимир Ильич Головач в своей рецензии в «Мир ПК» одним из первых предсказал ей долгую жизнь. Андрей Андреевич Терехов, известный знаток компьютерной литературы, также высоко оценил качество книги. Очень хорошо отзывались о ней Владимир Арнольдович Биллиг, Руслан Петрович Богатырев, Лев Николаевич Чернышов, а также многие другие. Всем этим людям огромная благодарность за поддержку и стимулирование настоящего издания.

Книга оказалась также в перечне основной литературы, рекомендованной Минобразованием России для специальности 010200 «Прикладная математика и информатика».

Немало замечательных членов программисткого сообщества, упоминаемых в книге, многих из которых автор имел удовольствие знать лично и даже обсуждать с ними фрагменты книги или читавшегося в МГУ курса, за прошедшие годы покинули этот мир. Среди них Евгений Андреевич Жоголев, Александр Владимирович Замулин, Михаил Романович Шура-Бура, Владимир Федорович Турчин. Бесконечная им признательность за бесценный вклад в общее дело и светлая память.

Переиздание книги, несмотря на имеющиеся запросы, совсем недавно представлялось совершенно нереальным с учетом стопроцентной загрузки автора основной работой. Ведь электронная верстка книги оказалась утраченной в результате непростых пертурбаций бурных 90-х прошлого века.

К счастью, мир полон чудес. Одно из них – воскрешение полной электронной версии книги с помощью современных средств сканирования с печатного оригинала и последующего ручного редактирования. Эта огромная работа была выполнена целиком по собственной инициативе Ольгой Львовной Бондаренко. Как по волшебству, подоспело и предложение Дмитрия Алексеевича Мовчана переиздать книгу в «ДМК Пресс».

Автору оставалось только вычитать полученный от Ольги Львовны Word-документ и передать результат в «ДМК Пресс» в надежде помочь людям, желающим глубоко вникнуть в суть проблем, принципов и концепций современного программирования.

Удачи, глубокоуважаемый читатель!

В. Ш. Кауфман
12.04.10 Москва–Хельсинки

Предисловие

Эта книга возникла из курса лекций «Языки программирования», читаемого автором в МГУ. Стимулом для написания книги послужило отсутствие доступной литературы, в которой были бы систематически изложены, во-первых, ключевые принципы, концепции и понятия, составляющие основу предмета и поэтому претендующие на относительную стабильность, и, во-вторых, перспективные идеи и тенденции, помогающие ориентироваться в огромном и быстро меняющемся мире современных языков программирования (ЯП).

Автор столкнулся с немалыми проблемами, несмотря на то что становление современных ЯП происходило, можно сказать, на его глазах. Пришлось пережить и восторги от изобретения первых «языков высокого уровня», быть среди тех, кто увлекался их «усовершенствованием» и созданием первых трансляторов, опираясь только на здравый смысл и собственную смекалку, пришлось пережить и надежды на создание «универсального ЯП» объединенными усилиями международного программистского братства, и разочарования от бездарной траты сил и средств на бесперспективные начинания.

Когда в результате преодоления части этих проблем выяснилось, что удается существенно прояснить суть дела (частично отбирая, частично изобретая принципы, концепции и понятия), фрагменты книги, к удивлению автора, оказались интересны не только студентам и коллегам-преподавателям, но и программистам – профессионалам и специалистам по ЯП. По-видимому, проблемы, с которыми столкнулся автор, осмысливая один из важнейших аспектов информатики, оказались жизненно важными для существенно более широкого круга потенциальных читателей, а отражение опыта их преодоления в тексте книги – достаточно интересным и поучительным.

Заметив это обстоятельство, автор уже сознательно стал иногда рассчитывать не только на студенческую аудиторию, но и на более искушенного читателя, позволяя себе намеки и аналогии, подразумевающие личный опыт программирования и даже экспертной деятельности в области ЯП. Более того, стало очень трудно отделить то, что известно, признано, устоялось, от того, что удалось только что понять, систематизировать, придумать. В результате жанр книги стал менее определенным, «поплыл» от первоначально задуманного учебного пособия в сторону монографии.

С точки зрения ключевых концепций и принципов, определяющих современное состояние и перспективы в области ЯП, конкретные ЯП интересны не сами по себе, а прежде всего как источники примеров при обсуждении излагаемых положений. Поэтому систематически применяется метод моделирования ЯП – изучаются не ЯП в целом, а только их модели. Конечно, автор старался дать необходимый минимум сведений о ЯП, позволяющий понимать написанные на нем примеры без привлечения дополнительной литературы. В качестве основного метода знакомства с ЯП в книге принят метод «погружения», столь популярный при ускоренном обучении иностранным языкам, – сведения о ЯП читателю предлагается извлекать непосредственно из примеров, написанных на этом ЯП программ (естественно, с подробными комментариями). Опыт показывает, что такой путь обеспечивает достижение основной цели с приемлемыми затратами времени и сил. Поэтому в книге нет подробных описаний конкретных ЯП – желающие могут воспользоваться официальными сообщениями, фирменной документацией или учебниками по ЯП.

Немыслимо в одной книге содержательно обсудить все (даже только важнейшие) концепции и принципы, определяющие современные ЯП. Пришлось выработать критерий отбора. Он касается и проблем программирования в целом, и назначения ЯП, и их выбираемых для обсуждения свойств. Из всех проблем программирования в качестве ключевой выбрана проблема сложности (самих программ, их создания, средств их создания и т. п.). Основным источником сложности считается семантический разрыв – рассогласование моделей мира у потенциального пользователя и потенциального исполнителя программ (компьютера). В качестве основных средств преодоления этого разрыва выделены, с одной стороны, аппарат абстракции-конкретизации (аппарат развития), а с другой – аппарат прогнозирования-контроля (аппарат защиты). Основной объект изучения – это концепции, принципы и понятия, позволяющие строить концептуально-целостные ЯП с мощным аппаратом развития и надежной защитой.

Книга состоит из двух частей. Первая посвящена основным абстракциям, используемым в современных ЯП. В качестве основного языка примеров здесь фигурирует ЯП Ада. Он удобен в этой роли потому, что в той или иной форме содержит ответы практически на все технологические проблемы. Другими словами, Ада служит примером «максимального» современного ЯП. «Минимальные» ЯП представлены языками Никлауса Вирта – это Модула-2 и Оберон (образца 1988 г.).

Вторая часть рассказывает о перспективных тенденциях в ЯП. В ней рассмотрены ситуационное, функциональное, доказательное, реляционное, параллельное и объектно-ориентированное программирование. Среди языков-примеров – Рефал, функциональный язык Бэкуса, Оккам-2 для программирования транспьютеров, обьектно-ориентированный Турбо Паскаль и др.

В книге немало вопросов и упражнений (снабженных обычно подсказками), призванных помочь читателю управлять своим вниманием и контролировать уровень усвоения материала. Результаты упражнений при дальнейшем изложении не используются.

Так как замысел книги возник восемь лет назад и почти половина материала написана еще в 1983–1985 гг., закономерно опасение, не устарела ли книга еще до своего выхода в свет. Конечно, судить об этом читателю, однако автор старался отбирать фундаментальные и, по его мнению, перспективные концепции и принципы, которые по самой своей природе должны быть стабильнее быстро меняющейся конъюнктуры.

Косвенным подтверждением такой стабильности послужил весьма обрадовавший автора факт, что современный всплеск (своеобразный «бум») интереса к объектно-ориентированному программированию – это в сущности всплеск интереса к средствам программирования, обеспечивающим рациональное развитие программных услуг при надежной защите авторского права. Но именно средства развития и защиты в ЯП и были выбраны в качестве самого интересного аспекта ЯП еще в начале работы над книгой. Такое знаменательное совпадение придает уверенности в правильности выбора и позволяет считать объектно-ориентированное программирование не просто очередной модой, а естественной «закрывающей скобкой» как очередного этапа в осознании системы ценностей в программировании, так и нашего рассмотрения концепций и принципов ЯП.

Еще одним подтверждением тезиса о фундаментальности рассматриваемых в книге концепций и принципов может служить тот факт, что как в разработанных в конце

1990 г. требованиях на создание обновленного международного стандарта языка программирования Ада (учитывающих самые современные пожелания интернационального сообщества пользователей языка Ада и самые современные методы их удовлетворения), так и в аванпроекте обновленного языка, датированном февралем 1991 г., нашли отражение, кроме объектно-ориентированного программирования, и такие рассмотренные в книге проблемы, как развитие концепции управления асинхронными процессами, развитие концепции типа, развитие концепции исключений и др.

Создавать эту книгу помогали многие люди, которые, конечно, не несут какой-либо ответственности за ее недостатки. Автору приятно выразить признательность В. К. Мережкову за инициативу и огромную помощь при издании первой части рукописи в НПО «Центрпрограммсистем», профессорам Е. Л. Ющенко, М. Р. Шура-Буре, В. Н. Редько, И. М. Витенбергу, А. А. Красилову, С. С. Лаврову, Я. Я. Осису, Е. А. Жоголеву, Н. П. Трифонову, Г. С. Цейтину за поддержку и ценные советы, своим коллегам и первым читателям В. Л. Темову, В. Н. Агафонову, В. И. Головачу, А. С. Маркову, Б. Г. Чеблакову, Анд. В. Климову, В. Н. Лукину, И. В. Раковскому за содержательную критику, А. Л. Александрову, И. Н. Зейтленок, И. З. Луговой и особенно С. И. Рыбину за помощь в подготовке рукописи, своим слушателям и студентам за внимание, терпение и любознательность, своим родным – за понимание и заботу.

Автор старался не изменить духу преданности сути дела и творческой раскованности, воплощением которых для него остаются рано ушедшие из жизни Андрей Петрович Ершов, успевший прочитать первый вариант рукописи и поддержать настроение автора писать «как пишется», и Адольф Львович Фуксман, который в свое время горячо обсуждал с В. Л. Темовым и автором совместный проект университетского учебника по программированию.

В. Ш. Кауфман

Часть I

Современное состояние языков программирования

Глава 1. Концептуальная схема языка программирования 21
Глава 2. Пример современного базового ЯП (модель А) 43
Глава 3. Важнейшие абстракции: данные, операции, связывание ... 69
Глава 4. Данные и типы 85
Глава 5. Раздельная компиляция 145
Глава 6. Асинхронные процессы 151
Глава 7. Нотация 175
Глава 8. Исключения 183
Глава 9. Библиотека 201
Глава 10. Именование и видимость (на примере Ады) .. 205
Глава 11. Обмен с внешней средой 219
Глава 12. Два альтернативных принципа создания ЯП 237

Глава 1

Концептуальная схема языка программирования

1.1. Что такое язык программирования 22
1.2. Метауровень 22
1.3. Модель передачи сообщения 23
1.4. Классификация недоразумений 23
1.5. Отступление об абстракции-конкретизации. Понятие модели 25
1.6. Синтактика, семантика, прагматика 26
1.7. Зачем могут понадобиться знания о ЯП 27
1.8. Принцип моделирования ЯП 29
1.9. Пять основных позиций рассмотрения ЯП 29
1.10. Что такое производство программных услуг 30
1.11. Производство программных услуг – основная цель программирования 32
1.12. Сложность как основная проблема программирования 33
1.13. Источники сложности 34
1.14. Два основных средства борьбы со сложностью. Основной критерий качества ЯП 36
1.15. Язык программирования как знаковая система 37
1.16. Разновидности программирования 38
1.17. Понятие о базовом языке 39
1.18. Концептуальная схема рассмотрения ЯП 40

1.1. Что такое язык программирования

Для начала дадим экстенсиональное определение ЯП – явно перечислим те конкретные языки, которые нас заведомо интересуют (их мы уверенно считаем языками программирования). Это Фортран, Симула, Паскаль, Бейсик, Лисп, Форт, Рефал, Ада, Си, Оккам, Оберон. Однако хочется иметь возможность на основе определения предсказывать новые частные случаи, в определении не перечисленные. Такое определение должно опираться на существенные свойства выбираемых для изучения языков – оно должно быть интенсиональным. Дадим одно из возможных интенсиональных определений ЯП.

Язык программирования – это инструмент для планирования поведения исполнителя.

Однако, во-первых, перечисленные выше ЯП служат не только для планирования поведения исполнителей (компьютеров), но и для обмена программами между людьми. Такая важнейшая функция существенно влияет на устройство и принципы создания ЯП (хотя она все же вторична – можно показать, что люди должны понимать и читать программы, даже не имея никаких намерений ими обмениваться; просто иначе достаточно крупной программы не создать). Эту функцию языка никак нельзя игнорировать при изучении ЯП.

Во-вторых, в нашем определении каждое слово нуждается в уточнении. Являются ли «инструментами для планирования поведения исполнителя» должностная инструкция, письменный стол, переговорное устройство, правила уличного движения, русский язык?

1.2. Метауровень

Взглянем на наши действия с позиции стороннего наблюдателя, отвлечемся от своей роли соответственно автора и читателей на только что законченном начальном отрезке нашей совместной работы. Другими словами, поднимемся на метауровень, чтобы обозревать исходный уровень в целом.

Чем мы занимались?

Во-первых, попытались добиться взаимопонимания в вопросе о том, что такое ЯП. Во-вторых, начали применять для достижения взаимопонимания метод последовательных уточнений.

Чего мы добились и что осталось неясным? Стало яснее, что будем изучать, – можем привести примеры ЯП, с которыми все согласны, и указать объекты, заведомо не являющиеся ЯП в соответствии с нашим определением (также рассчитывая на общее согласие), скажем, левая тумба письменного стола. Почувствовали, что добиться взаимопонимания (даже по поводу привычных понятий) очень непросто. Осталось неясным, в частности, с какой позиции и с какой целью мы намерены изучать ЯП.

Постараемся в первом приближении устранить эти неясности. Однако заниматься последовательными уточнениями многих важных понятий мы будем на

протяжении всей нашей работы – ведь она не формальная, а содержательная, нас интересуют реально существующие, возникающие на наших глазах и развивающиеся объекты – живые ЯП. Поэтому-то и невозможно дать исчерпывающего описания ЯП как понятия (это понятие живет вместе с нами).

Начнем с более внимательного рассмотрения преград, обычно возникающих на пути к взаимопониманию.

1.3. Модель передачи сообщения

Добиться взаимопонимания бывает очень сложно. Чтобы выделить возникающие здесь проблемы, рассмотрим следующую модель передачи сообщения (рис. 1.1).

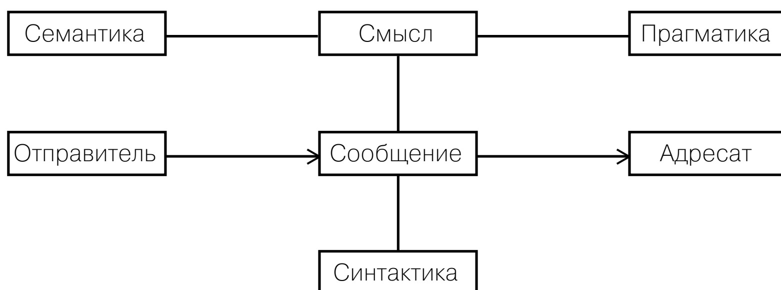


Рис. 1.1

В этой модели выделены понятия «отправитель» (автор, генератор сообщения), «адресат» (получатель, читатель, слушатель сообщения), собственно «сообщение» (текст, последовательность звуков), «смысл» сообщения (нечто обозначаемое сообщением в соответствии с правилами, известными и отправителю, и адресату).

Выделены также названия наук, занимающихся соответственно правилами построения допустимых сообщений (синтактика), правилами сопоставления таким сообщениям смысла (семантика) и правилами, регулирующими использование сообщений (прагматика).

1.4. Классификация недоразумений

С помощью модели на рис. 1.1 займемся классификацией недоразумений, возникающих при попытке установить взаимопонимание.

Автор (отправитель сообщения) может подразумевать одну структуру сообщения, а адресат (получатель) – другую, как в классическом королевском указе: «Казнить нельзя помиловать!» Это синтаксическое недоразумение.

Автор может употребить слово с неточным смыслом, подразумевая один его оттенок, а адресат выберет другой. Рассмотрим, например, фрагмент рецепта при-

готовления шоколадной помадки: «изредка помешивая, варить на слабом огне до тех пор, пока капля не станет превращаться в холодной воде в мягкий шарик». Не потому ли кулинарное искусство и является искусством, а не точной наукой, что разные повара, зная один и тот же рецепт, по-разному понимают слова «изредка», «медленно», «холодной», «мягкий», а также с разной частотой станут пробовать «превратить каплю в мягкий шарик». Естественно, и результаты у них будут разные. Это семантическое недоразумение.

Наконец, автору трудно иногда представить себе, какую интерпретацию может придать его сообщению адресат, если у них сильно различаются представления о мире или решаемые задачи.

Например, сообщение лектора о предстоящем коллоквиуме может быть воспринято студентами как призыв не посещать малоинформативные лекции, чтобы иметь время для работы с книгами. Это уже прагматическое недоразумение.

Нетрудно привести и другие примеры синтаксических, семантических и прагматических недоразумений при попытке достичь взаимопонимания.

Почему же в самом начале речь пошла о взаимопонимании и о стоящих на пути к нему преградах? В основном по двум причинам.

Во-первых, ЯП – это инструмент для достижения взаимопонимания (безопаснее «взаимопонимания») людей с компьютерами и между людьми по поводу управления компьютерами. Поэтому в принципах построения, структуре, понятиях и конструктах ЯП находят свое отражение и сущность общей проблемы взаимопонимания, и взгляды творцов ЯП на эту проблему, и конкретные методы ее решения.

Во-вторых, способ, которым люди преодолевают преграды на пути к взаимопониманию, содержит некоторые существенные элементы, остающиеся важными и при общении с компьютерами (в частности, при создании и использовании ЯП). Кстати, в Международной организации по стандартизации ИСО разработан документ [1], регламентирующий устройство стандартов ЯП. Он содержит классификацию программных дефектов, полностью согласующуюся с нашей классификацией недоразумений.

Особенно бояться синтаксических недоразумений не стоит. Они касаются отдельных неудачных фраз и легко устраняются немедленным вопросом (устным или письменным). В ЯП это тоже не проблема – таких недоразумений там просто не бывает. Дело в том, что создатели ЯП руководствуются принципом однозначности: **язык программирования должен быть синтаксически однозначным** (то есть всякий правильный текст на ЯП должен иметь единственную допустимую структуру).

Итак, сформулирован один из принципов построения ЯП, отличающих их, например, от языков естественных. Такого рода общие принципы и концепции нас и будут интересовать в первую очередь.

Семантические недоразумения опаснее. Если, скажем, слово «язык» будет ассоциироваться с субпродуктом, ставшим весьма редким гостем прилавка, то недоразумение может не ограничиться пределами одной фразы. Большой язык, свежий язык, красный язык, зеленый и голубой язык – все это может касаться и го-

вяжьего языка, и ЯП (в конкурсе языковых проектов, ставшем одним из этапов создания языка Ада, языки-конкуренты получили условные «цветные» наименования; победил «зеленый» язык).

Метод борьбы с семантическими недоразумениями при человеческом общении известен – нужно выделять важные понятия, давать им четкие определения, приводить характерные примеры. Это со стороны говорящего. Слушатели должны, в свою очередь, стараться уловить оставшиеся существенные неясности, приводить контрпримеры (объектов, соответствующих определениям, но, по-видимому, не имевшихся в виду говорящим, и объектов, не соответствующих определениям, но, скорее всего, имевшихся в виду). Этот же метод точных определений широко используется в ЯП (определения процедур, функций и типов в Паскале), а примеры и контрпримеры применяются, как известно, при отладке программ.

1.5. Отступление об абстракции-конкретизации. Понятие модели

Добиваясь взаимопонимания, мы активно пользуемся аппаратом абстракции-конкретизации (обобщения-специализации).

Создавая понятие, отвлекаемся (абстрагируемся) от несущественных свойств тех конкретных объектов, на основе знания которых понятие создается, и фиксируем в создаваемом понятии лишь свойства существенные, важные с точки зрения задачи, решаемой с применением этого понятия. Так, в понятии «часы» мы обычно фиксируем лишь свойство быть «устройством, показывающим время», и отвлекаемся от формы, структуры, цвета, материала, изготовителя и других атрибутов конкретных часов.

Приводя пример, мы конкретизируем абстрактное понятие, «снабжая» его второстепенными с точки зрения его сущности, но важными в конкретной ситуации деталями. Так, конкретное выполнение процедуры происходит при конкретных значениях ее параметров; у конкретного примера ЯП – Фортрана – конкретный синтаксис и конкретная семантика.

Мои часы большие, круглые, позолоченные, с тремя стрелками, марки «Восток», на 17 камнях. Все это немного говорит об их качестве в роли «устройства, показывающего время», но конкретное устройство всегда обладает подобными «необязательными», с точки зрения его роли, свойствами. Их существование лишь подчеркивает тот факт, что (абстрактное) понятие никогда не исчерпывает конкретного объекта – оно всегда отражает лишь некоторую точку зрения на этот объект, служит компонентой его модели, оказавшейся удобной для решения определенной задачи. В другой ситуации, при решении другой задачи, этот же конкретный объект может играть другую роль. Тогда и точка зрения на него может быть другой, и может потребоваться совсем другая модель того же самого объекта.

На «устройство, показывающее время», в известных условиях можно предпочесть смотреть как на «украшение», и с этой точки зрения (в этой его роли) важнее станут форма, цвет, размер, фирма, важнее даже способности правильно пока-

зывать время. На процедуру можно смотреть как на «объект, расходующий машинные ресурсы». При такой ее роли совершенно не важно, каков смысл выполняемых в ней действий. На ЯП иногда приходится смотреть как на объект стандартизации, и тогда важно не столько то, каковы именно особенности его семантики и синтаксиса, сколько то, найдется ли достаточно много заинтересованных в тех или иных его свойствах людей и организаций.

Непроизвольная, а иногда и намеренная, но не подчеркнутая явно смена точки зрения, переход по существу к другой модели объекта мешает взаимопониманию, служит источником прагматических недоразумений. Вы говорите, что часы «плохие», потому что некрасивые, а я говорю «хорошие», так как они отлично работают.

Способность без затруднений переходить от одной модели к другой, четко фиксировать и легко изменять уровень рассмотрения, а также угол зрения, отмечается обычно как важнейшее профессиональное качество программиста.

1.6. Синтактика, семантика, прагматика

Устранять прагматические недоразумения бывает особенно сложно, когда они связаны с различием не только точек зрения, но и целевых установок. Если правильно разложить фразу на составляющие может помочь согласование с контекстом, а правильно понять смысл слова или фразы может помочь знание их назначения (роли), то восстановить эту роль, догадаться о ней, если об этом не сказано явно, очень тяжело. Слишком велика неопределенность, свобода выбора.

Представим себе положение человека, которому излагается последовательность определений и не говорится, зачем они вводятся, для решения каких задач предназначены. Это хорошо знакомая всем ситуация – есть такой стиль изложения математических результатов. Слушатель (читатель) при этом лишен всякой опоры для контроля, кроме поиска чисто логических противоречий. Пока он не понял, зачем все это нужно, он может пропустить любую содержательную ошибку. А попробуйте понять смысл программы, если неизвестно, для чего она написана!

Вывод очевиден – для достижения взаимопонимания необходимо, чтобы отправитель и адресат пользовались, во-первых, одинаковыми правилами разложения сообщения на составляющие (изучением таких правил занимается синтактика); во-вторых, согласованными правилами сопоставления сообщению смысла (такими правилами занимается семантика); в-третьих, имели согласованные целевые установки (это предмет прагматики).

Ролью перечисленных аспектов для создания и использования ЯП мы еще займемся, а сейчас уместно поговорить об основной цели книги (для согласования наших целевых установок).

Мы намерены изложить принципы оценки, создания и использования современных ЯП. Это очень нужная, плодотворная и увлекательная, но далеко не устоявшаяся, быстро развивающаяся область. Поэтому нет возможности опираться на освященный традицией опыт предшественников, а также стабильные программы

и учебники (как это бывает, скажем, при изучении математического анализа или дифференциальных уравнений). Приходится рисковать и экспериментировать.

Итак, о нашей основной цели. Она состоит в том, чтобы постараться правильно ориентировать читателя в области ЯП, помочь ему осознать навыки и опыт, приобретенные при самостоятельной работе с конкретными ЯП.

Но не слишком ли опасна идея «правильно» ориентировать? Ведь если, скажем, представления автора о профессиональных запросах читателя или о тенденциях развития ЯП окажутся ошибочными, то скорее всего «правильная» ориентация на самом деле окажется дезориентацией. Не лучше ли ограничиться изложением бесспорных положений из области ЯП – уж они-то понадобятся наверняка?!

К сожалению или к счастью, альтернативы у нас, по сути, нет. Абсолютно бесспорные положения касаются, как правило, лишь конкретных ЯП. Например, «Один из операторов в языке Паскаль – оператор присваивания. Устроен он так-то. Служит для того-то». В хорошо известном учебнике программирования это положение обобщено. Сказано так: «Фундаментальным действием в любом алгоритмическом языке является присваивание, которое изменяет значение некоторой переменной». И это уже неверно! Сейчас много внимания уделяется так называемому функциональному программированию, аппликативным ЯП, где присваивание – не только не «фундаментальное» действие, но его вообще нет!

Значит, в области ЯП нет достаточно общих бесспорных положений? В некотором смысле есть. Чаще не столь бесспорных, сколь заслуживающих изучения. Правда, их общность несколько другого характера. Примером может служить упоминавшийся принцип однозначности. Да и приведенная фраза из учебника – вполне бесспорное положение, если считать, что она характеризует определенный класс ЯП, в который не попадает, скажем, язык Лисп – один из самых «заслуженных», распространенных и в то же время перспективных. Итак, даже если ограничиться лишь относительно бесспорными положениями, их все равно нужно отбирать с определенных позиций, с определенной целью. Естественная цель – стремиться принести читателю максимальную пользу. Опять мы приходим к «угадыванию» будущих потребностей.

1.7. Зачем могут понадобиться знания о ЯП

Во-первых, каждая программа должна общаться (обмениваться информацией) с внешним миром. Соглашения, определяющие способ общения, – это язык, так что понимание принципов построения языков – необходимая компонента грамотного программирования. Исключительно важная компонента, потому что непосредственно связана с внешним эффектом программы, со способом ее использования. При разработке внешнего сопряжения своей программы программист обязан проявить истинный профессионализм, предоставляя пользователю максимум услуг при минимуме затрат. Особенно это важно при создании пакетов приклад-

ных программ, инструментальных систем, вообще любых программных изделий, предназначенных для эксплуатации без участия автора.

Во-вторых, каждый ЯП – это своя философия, свой взгляд на деятельность программиста, отражение определенной технологии программирования. Даже представлений об Алголе-60, Фортране и Бейсике достаточно, чтобы почувствовать, что имеется в виду.

Скажем, творцы Алгола (выдающиеся представители международного сообщества ученых в области информатики под руководством Петера Наура) с естественным для них академизмом придавали относительно много значения строгости определения и изяществу языковых конструктов. Считалось, что самое важное в работе программиста – сформулировать алгоритм (и, возможно, опубликовать его). Переписать программу в расчете на конкретные устройства ввода-вывода считалось не заслуживающей особого внимания технической деятельностью. Не привлек должного внимания авторов языка и такой «технический» аспект программистской деятельности, как компоновка программ из модулей.

Творцы Фортрана (сотрудники фирмы IBM во главе с Джоном Бэкусом) в значительной степени пренебрегли строгостью и изяществом языка и со свойственным им в ту пору (1954–1957 гг.) прагматизмом уже в первых версиях языка уделили особое внимание вводу-выводу и модульности. Но ни Фортран, ни Алгол не рассчитаны на работу в диалоговом режиме, в отличие от Бейсика (созданного в Дартмундском колледже первоначально для обучения студентов).

Таким образом, изучение ЯП дает знание и понимание разнообразных подходов к программированию. Это полезно при любой программистской деятельности.

В-третьих, понимание общих принципов и концепций, определяющих строение и применение ЯП, позволяет легче и глубже освоить конкретный язык – основной профессиональный инструмент программиста.

В-четвертых, и это хотелось бы подчеркнуть особо, понятия и тенденции в области ЯП с некоторым запаздыванием (в целом полезным) довольно точно отражают понятия и тенденции собственно программирования как науки, искусства и ремесла (и просто области человеческой деятельности). В этом смысле мы обсуждаем основные принципы и понятия программирования, но со специфически языковой точки зрения.

Все, о чем было сказано до сих пор, касалось интересов потенциального пользователя ЯП. Но читатель может оказаться и руководителем коллектива, которому требуется оценивать и выбирать ЯП для выполнения конкретного проекта (учитывать, скажем, затраты на освоение этого ЯП или на обмен написанными на нем программными изделиями). Если же он станет творцом ЯП, создателем транслятора или руководства для пользователей, то ему понадобятся столь разнообразные знания о ЯП, что их придется извлекать целеустремленным изучением специальной литературы. Можно надеяться дать лишь первоначальный импульс в нужном направлении.

Конечно, предсказать, для чего именно понадобятся приобретенные знания, сложно. Могут напрямую и вовсе не понадобиться. Но наверняка пригодится приобретенная обсуждениями, размышлениями и упражнениями культура рабо-

ты со сложными объектами при решении сложных задач. В нашем случае это такие задачи, как оценка, использование, разработка и реализация ЯП.

1.8. Принцип моделирования ЯП

Было бы неправильно ставить нашей целью научить свободному владению конкретными ЯП, пусть даже особо привлекательными или перспективными. Для этого служат специальные учебники, упражнения и, главное, практика.

Наша задача – познакомить с важнейшими понятиями и концепциями, помогающими оценивать, использовать, реализовывать и разрабатывать ЯП, дать представление о направлениях и проблемах их развития. Поэтому займемся в основном изучением моделей ЯП. Другими словами, при изучении ЯП будем систематически применять принцип моделирования (как самих реальных ЯП, так и их отдельных аспектов). Наиболее важные по тем или иным причинам ЯП или их конструкты иногда будут рассмотрены довольно подробно, но прежде всего лишь как примеры, иллюстрирующие более общие положения.

Например, важно понимать, что с каждым ЯП связан эталонный (абстрактный) исполнитель, в котором, в свою очередь, определены данные, операции, связывание, именование, аппарат прогнозирования и контроля, а возможно, и аппарат исключений, синхронизации и защиты. Важно понимать перечисленные термины, назначение соответствующих языковых конструктов и уметь ими пользоваться при решении практических задач. Но не очень важно помнить наизусть все связанные с ними тонкости в конкретных ЯП. Последнее может оказаться важным лишь тогда, когда тонкости иллюстрируют ключевые концепции рассматриваемого ЯП. Например, жесткие правила выбора обозначений в Бейсике непосредственно связаны с его ориентацией на относительно небольшие программы и простоту реализации.

1.9. Пять основных позиций рассмотрения ЯП

Итак, будем считать, что целевые установки согласованы в достаточной степени, чтобы сделать следующий шаг – приступить к систематическому изучению нашего предмета.

И сразу вопрос – с чего начать? Легко сказать «систематическому». Но ведь системы бывают разные. Часто начинают «снизу» – с основных конструктов, встречающихся почти во всех существующих ЯП. Тогда мы сразу погружаемся в мир переменных, констант, параметров, процедур, циклов и т. п. Такой путь привлекателен хотя бы тем, что им сравнительно легко пойти. Но на этом пути за деревьями обычно не видно леса, не удается увидеть ЯП в целом, построить его адекватную модель.

Поэтому выберем другой путь. Постараемся взглянуть на объект нашего изучения – ЯП – с общих позиций. Нас будут особенно интересовать технологическая, семиотическая и авторская позиции.

Первая названа технологической потому, что отражает взгляд человека, желающего или вынужденного пользоваться ЯП как технологическим инструментом на каком-либо из этапов создания и использования программных изделий (другими словами, в течение их жизненного цикла). С таким человеком естественно объясняться в технологических терминах.

Вторая позиция названа семиотической потому, что ее можно представить себе как позицию человека, знакомого с некоторыми знаковыми системами (русским языком, дорожными знаками, позиционными системами счисления) и желающего узнать, чем выделяются такие знаковые системы, как ЯП. Следует объяснить ему это в семиотических терминах.

Третья позиция – авторская. Автор создает ЯП, делает его известным программистской общественности, исправляет и модифицирует его с учетом поступающих предложений и критических замечаний.

Уделим внимание и другим позициям – математической и реализаторской.

Математик понимает, что такое математическая модель изучаемого объекта, и желает познакомиться с математическими моделями ЯП. С ним желательно объясняться в математических терминах.

Реализатор обеспечивает возможность пользоваться ЯП как средством практического программирования. Другими словами, он не только создает трансляторы, но и пишет методические руководства, обучающие и контролирующие программы, испытывает трансляторы и т. п.

Уместно подчеркнуть, что с разных позиций мы будем рассматривать один и тот же объект. Начнем с технологической позиции. Установим связь ЯП с производством программных услуг.

1.10. Что такое производство программных услуг

Напомним исходные понятия, известные из общего курса программирования компьютеров. Понятие компьютер нужно уточнить лишь в той мере, в которой это необходимо для нашей цели. Важно, что компьютер обладает двумя фундаментальными способностями – хранить данные и выполнять планы.

Начнем со второй способности. Назовем исполнителем всякое устройство, способное выполнять план. Так что и компьютер, и робот, и рабочий, и солдат, и сеть компьютеров, и коллектив института способны играть роль исполнителя.

Всего одна фундаментальная способность – выполнять план – дает возможность исполнителю предоставить пользователю содержательно разнообразные услуги. Определяя конкретные планы, можно настраивать исполнителя на предоставление конкретных услуг. Важно лишь, чтобы в плане фигурировали задания, посильные для выбранного исполнителя. Посильные – это значит такие, для выполнения которых у исполнителя имеются соответствующие ресурсы.

Для компьютеров как исполнителей характерны два вида ресурсов – память и процессор. Память реализует первую из двух названных фундаментальных способностей – служит для хранения данных. Это пассивный ресурс. Процессор реа-

лизует вторую из названных способностей – служит для выполнения действий, предусмотренных в планах. Это активный ресурс. Процессор характеризуется определенным набором допустимых действий (операций, команд). Действия из этого набора считаются элементарными (в плане не нужно заботиться о способе выполнения таких действий).

Две названные способности связаны – выполнение достаточно сложного плана требует его хранения в доступной исполнителю памяти. В свою очередь, реализация хранения требует способности выполнять план (данные нужно размещать, перемещать, делать доступными).

План для такого исполнителя, как компьютер, должен в итоге сводиться к указанию конечной последовательности элементарных действий. Такой план называют программой.

Людей как исполнителей характеризует прежде всего наличие у них модели реального мира, в достаточной степени согласованной с моделью мира у создателя плана. Поэтому в плане для людей можно указывать цели, а не элементарные действия.

Ресурс, существенный почти для всех реальных исполнителей, – это время. Важное свойство компьютеров как исполнителей – способность выполнять элементарные действия исключительно быстро (порядка микросекунды на действие). Не менее важное свойство компьютера – способность хранить огромные объемы данных (в оперативной памяти – мегабайты; на внешней – практически не ограничено).

Именно способность компьютеров выполнять весьма длинные последовательности элементарных действий над данными любого нужного объема за практически приемлемое время предоставляет пользователям весьма разнообразные по содержанию услуги (развлекать, играя с ним в интеллектуальные и (или) азартные игры, давать справки, помогать в составлении планов, управлять самолетами и танками, поддерживать светскую беседу).

Чтобы настроить компьютер на конкретный вид услуг, нужно снабдить его соответствующими этому виду услуг знаниями в приемлемой для него форме. Принципиально важный для нас факт, ставший очевидным лишь относительно недавно (по мере расширения и развития сферы предоставляемых компьютерами услуг), состоит в том, что самое сложное (и дорогое) в этом деле – не сами компьютеры, а именно представление знаний в приемлемой для них форме. В наше время аппаратура в компьютере по своей относительной стоимости иногда сравнима с упаковкой, в которую «заворачивают» знания.

Знания, представленные в компьютерах, традиционно называют программами (таким образом, под программой в широком смысле слова понимают не только планы). Соотношение стоимости аппаратуры и программ (а также иные соображения) во многих случаях дает основание абстрагироваться от используемой аппаратуры и говорить об услугах, предоставляемых непосредственно программами, а также о производстве программных услуг.

Подчеркнем, что пока далеко до полной независимости программ от используемой аппаратуры. Проблема переноса программ (на исполнитель другого типа) – одна из острейших проблем современного программирования. И тем не ме-

нее обычно затраты на создание достаточно сложной программы определяются в основном сущностью решаемой задачи (предоставляемой услуги), а не используемой аппаратуры.

Правомерность абстракции от аппаратуры подчеркивает определенную искусственность проблемы переноса программ. Если пока в значительной степени безразлично, на чем программировать, то разнообразие и несовместимость исполнителей вызваны не объективными, а социальными причинами.

Знания, представляемые в компьютерах, можно разделить на пассивные и активные. Первые отражают факты, связи и соотношения, касающиеся определенного вида услуг. Вторые – это рецепты, планы действий, процедуры, непосредственно управляющие исполнителем.

Представление пассивного знания ориентировано в первую очередь на такой ресурс компьютера, как память, а представление активного – на процессор. Исторически самыми первыми сферами применения компьютеров оказались такие, где главенствовало активное знание – эксплуатировалась способность ЭВМ не столько много знать, сколько быстро считать.

Кстати, и память на первых ЭВМ была очень мала по сравнению со скоростью работы процессора. Всю свою память они могли просмотреть (или переписать) за десятую долю секунды. Недалеко ушли в этом отношении и современные компьютеры (если говорить об оперативной памяти).

Соотношение между объемом оперативной памяти и скоростью процессоров активно влияет на мышление программистов, инженеров, пользователей, а также всех связанных с компьютерами людей. Изменение этого соотношения (а его разумно ожидать) способно произвести революцию в практическом программировании (см. далее о модифицированной модели Маркова и функциональном стиле программирования (модель Бэкуса); есть и другие перспективные модели, например реляционная). Пока же программисты вовсю экономят память, помещая новые значения на место старых, а преподаватели учат искусству такого «эффективного» программирования. В «эффективных» программах трудно восстановить процесс получения результата, трудно объяснить неожиданный результат. Традиционный стиль программирования, обусловленный бедностью ресурсов, затрудняет написание, понимание, проверку и удостоверение правильности программ. Тенденция развития состоит в том, что роли активного и пассивного знаний в производстве программных услуг становятся более симметричными, а забота об эффективности отступает перед заботой о дружественности программы.

1.11. Производство программных услуг – основная цель программирования

Измерять эффективность того или иного производства разумно лишь по отношению к его цели (конечному результату). Поэтому важно понимать, что конечная цель программирования – не создание программ самих по себе, а предоставление

программных услуг. Другими словами, программирование в конечном итоге нацелено на обслуживание пользователей. А настоящее обслуживание должно руководствоваться известным принципом: «Клиент всегда прав». В применении к программированию этот принцип означает, что программы должны быть «дружественными» по отношению к пользователю. В частности, они должны быть надежными, робастными и «заботливыми».

Первое означает, что в программе должно быть мало ошибок, второе – что она должна сохранять работоспособность в неблагоприятных условиях эксплуатации, третье – что она должна уметь объяснять свои действия, а также ошибки пользователя.

Что значит «мало ошибок», зависит от назначения программы (ясно, что программа обучения русскому языку и программа автопилота могут иметь различную надежность). Под «неблагоприятными условиями» понимается ограниченность выделяемых программе ресурсов (память, каналы ввода-вывода, число процессоров), перегрузка (много пользователей, большие объемы данных), ошибки пользователей, сбои и отказы аппаратуры, попытки намеренно вывести программу из строя и т. п.

Сказанное относится к работе программы. Однако важно понимать, что программная услуга – это не только услуга, оказываемая пользователю непосредственно при работе компьютера под управлением программы, но и проверка, оценка, продажа, подбор программ, их перенос на другой исполнитель, сопровождение и аттестация (авторитетное подтверждение качества) программ и т. п.

Когда производство программных услуг достигает некоторой степени зрелости, из кустарного производства вырастает индустрия программных услуг и обслуживающая ее потребности теория программирования. Как индустрия, так и кустарное производство пользуются при этом той или иной технологией – технологией производства программных услуг, то есть технологией программирования.

(О связи науки, искусства, теории и технологии в программировании см. замечательную Тьюринговскую лекцию **Дональда Кнута** в Communication of the ACM. – 1974. – Vol. 12. – P. 667–673).

1.12. Сложность как основная проблема программирования

Итак, основная цель программирования – производство программных услуг. Известно, что этот род человеческой деятельности в развитых странах уверенно выходит на первое место среди всех других производств (скажем, в США соответствующая отрасль хозяйства уже опередила недавних лидеров – нефтяную и автомобильную отрасли).

Вместе с тем известно, что создание программ и предоставление других связанных с ними услуг остается слишком дорогим и относительно длительным делом, в котором трудно гарантировать высококачественный конечный результат.

В чем же основная причина такого положения? Связана ли она с самой природой программирования или носит субъективный характер?

В настоящее время краткий ответ можно сформулировать так: «сложность – основная проблема программирования; связана с самой его природой; можно надеяться на ее понижение для освоенных классов задач».

1.13. Источники сложности

Попытаемся подробнее разобраться с тем, почему же сложность объектов, с которыми приходится иметь дело, – отличительная черта программирования компьютеров. Найдя источники сложности, можно с большей надеждой на успех искать пути ее преодоления.

Когда исполнители работают медленно, а действия, которые считаются элементарными, специфичны для вида предоставляемых услуг, то планирование деятельности исполнителя и критерии качества такого планирования существенно зависят и от услуги, и от конкретного набора элементарных действий.

Вряд ли стоит поручать, скажем, заводскому технологу, специалисту по обработке металлов резанием, планирование индивидуального пошива верхней одежды. Для этого есть закройщики, которые, в свою очередь, вряд ли смогут применить свое искусство при создании заводских технологических карт. Другими словами, набор «элементарных» действий двух рассмотренных категорий исполнителей считать эквивалентными неестественно.

Компьютеры работают быстро, и наборы их команд в известном смысле эквивалентны. Причем уже в течение многих лет сохраняется тенденция к увеличению скорости и объема памяти при фиксированной цене (примерно на порядок за десятилетие). В таких условиях возникает принципиальная возможность настроить исполнителя на предоставление услуг из очень широкого класса (во всяком случае, границы этого класса неизвестны). Для этого достаточно снабдить исполнителя подходящим планом (написать программу).

Эта принципиальная возможность соблазняет настраивать компьютеры на виды услуг, очень далеких от элементарных возможностей исполнителя. Более того, таковы почти все практически значимые услуги. Поэтому в общем случае план должен предусматривать огромное количество элементарных действий над огромным количеством элементарных объектов. Но этого мало. Самое главное – огромное количество связей между этими объектами. Поэтому и сами программы становятся огромными (уже имеются программы из миллионов команд, например для управления военными и космическими объектами).

Между тем способности человека работать с большим числом связанных объектов, как хорошо известно, весьма ограничены. В качестве ориентира при оценке этих способностей указывают обычно на так называемое «число Ингве», равное семи (плюс-минус 2). Другими словами, человек обычно не в состоянии уверенно работать с объектом, в котором более семи компонент с произвольными взаимными связями. До тех пор, пока программирование остается в основном человеческой деятельностью, с указанным ограничением необходимо считаться.

Таким образом, предоставив универсальность, скорость и потенциально неограниченную память, создатели компьютеров, с одной стороны, соблазнили человечество неслыханными возможностями, а с другой – поставили лицом к лицу с проблемами невиданной потенциальной сложности (при попытке осуществить эти гипотетические возможности).

В этой связи упомянем известный принцип «труд на юзера спихнуть» (*user* – пользователь). Изобретатели компьютеров, предоставив в распоряжение программистов исключительное по своим возможностям абстрактное устройство, «спихнули» на них труд по настройке этого абстрактного устройства на предоставление конкретных услуг. Но такая конкретизация оказалась далеко не тривиальной. Программисты, в свою очередь, создавая «универсальные» программы, «спихивают» труд по их применению в конкретных условиях на потенциальных пользователей этих программ.

Итак, первый источник сложности в программировании – так называемый семантический разрыв – разрыв между уровнем и характером элементарных объектов и операций, с одной стороны, и потенциально возможных услуг – с другой. Иными словами, это проблема согласования масштаба – ювелирными инструментами предлагается сооружать города.

Именно этот источник имелся в виду, когда шла речь об объективном характере присущей программированию сложности. Занимаясь определенным классом услуг (задач), можно стремиться выделить характерный именно для этого класса набор элементарных объектов и операций, построить соответствующий исполнитель (аппаратным или программным способом) и программировать на таком более подходящем исполнителе. Фактически это означает создать адекватный выбранному классу услуг ЯП. На практике это самый распространенный способ борьбы со сложностью и одновременно основная причина роста проблемно-ориентированных языков (ПОЯ).

Имеется еще один принципиально важный источник сложности, затрудняющий «взаимопонимание» с компьютерами. Речь идет об отсутствии в современных компьютерах модели реального мира, согласованной с представлениями о мире у программистов и пользователей. Поэтому в общем случае компьютер не в состоянии контролировать указания программиста или действия пользователя с прагматической точки зрения (контролировать соответствие между действиями и теми целями, ради которых эти действия совершаются, – цели компьютеру неизвестны).

Из-за этого самая «мелкая», с точки зрения создателя программы, описка может привести к совершенно непредсказуемым последствиям (широко известен случай, когда из-за одной запятой в программе на Фортране взорвалась космическая ракета, направлявшаяся на Венеру; пропали усилия, стоившие миллиарды долларов).

Итак, в качестве второго источника сложности в современном программировании следует назвать незнание компьютером реального мира. Лишенный необходимых знаний, компьютер не может не только скорректировать неточно указанные в программе действия, но и проинформировать об отклонениях от направления на

цель работы. Традиционное для компьютеров управление посредством указания действий, а не целей требует учета мельчайших нюансов всех обстоятельств, в которых может оказаться исполнитель в процессе предоставления нужной услуги. Это резко увеличивает число объектов, с которыми приходится иметь дело при создании программ. Отсюда повышение сложности программирования, увеличение размера программ, понижение их надежности и робастности.

Со вторым источником сложности борются, развивая методы представления в компьютерах знаний о реальном мире и эффективном учете этих знаний при создании и исполнении «дружественных» программ.

1.14. Два основных средства борьбы со сложностью. Основной критерий качества ЯП

Рассмотренные источники сложности оказывают определяющее влияние на теорию и практику в области ЯП. Важнейшим средством борьбы с семантическим разрывом служит аппарат абстракции-конкретизации, имеющийся в том или ином виде в любом ЯП. Именно этот аппарат – основа проблемной ориентации языковых выразительных средств.

Например, в Фортране характерное средство абстракции – подпрограмма, а соответствующее средство конкретизации – обращение к ней с фактическими параметрами. Поэтому естественный ПОЯ, создаваемый посредством Фортрана, – набор (пакет) проблемно-ориентированных подпрограмм. В более современных ЯП применяют более развитые средства абстракции (абстрактные типы данных, кластеры, фреймы) и соответствующие средства конкретизации (о которых мы еще поговорим). Становятся более богатыми и возможности строить ПОЯ.

Важнейшим средством борьбы с незнанием реального мира служит аппарат прогнозирования-контроля. Имеются ЯП, в которых этот аппарат практически отсутствует (Апл, Форт, Лисп) или очень слаб (любой ассемблер). Однако именно этот аппарат – основа повышения надежности и робастности программ. Последнее не означает, что «дружественные» программы невозможно писать на ЯП со слабым прогнозированием-контролем. Просто в этом случае создание подходящего аппарата полностью возлагается на программиста.

Например, в Фортране характерное средство прогнозирования – встроенные (предопределенные) типы данных. Соответствующий контроль предусмотрен семантикой языка, но средств управления таким контролем нет (новые типы данных вводить нельзя). В таких ЯП, как Паскаль или Ада, этот аппарат более развит, а в так называемых языках искусственного интеллекта он прямо предназначен для представления достаточно полных знаний о мире, целей деятельности и контроля действий как самой программы, так и пользователя.

Упражнение. Приведите примеры средств абстракции-конкретизации и прогнозирования-контроля в известных вам ЯП. Постарайтесь подобрать симметричные,

взаимно дополнительные средства. Убедитесь, что эта дополнительность обеспечена не всегда.

Теперь мы готовы сформулировать следующий основной критерий качества ЯП (как инструмента, то есть с технологической позиции): **язык тем лучше, чем менее сложно построенное на его основе производство программных услуг**.

На этом оставим пока технологическую позицию и займемся семиотической.

1.15. Язык программирования как знаковая система

Продолжим уточнение понятия «язык программирования». Наше новое интенсиональное определение ЯП таково:

язык программирования – это знаковая система для планирования поведения компьютеров.

Итак, не любой «инструмент», а «знаковая система», и не для произвольных «исполнителей», а только для компьютеров. К ограничению класса исполнителей в этом определении мы подготовились заранее, а вот о знаковых системах еще подробно не говорили.

Знаковая система – это совокупность соглашений (явных или неявных), определяющих класс знаковых ситуаций. Понятие знаковой ситуации в семиотике относят к первичным понятиям, представление о которых создают с помощью примеров, а не явных определений. Необходимые компоненты знаковой ситуации – знак и денотат. Говорят, что знак обозначает денотат (знак называют также обозначением, или именем, а денотат – обозначаемым, или значением). Так, в модели передачи сообщения само сообщение служит знаком, его смысл – денотатом.

Вот еще знаковые ситуации (первым укажем знак, вторым – денотат): буква и соответствующий звук, дорожный знак («кирпич») и соответствующее ограничение («въезд запрещен»), слово и соответствующее ему понятие. Каждый без затруднений пополнит этот список.

Когда класс знаковых ситуаций определяется совокупностью соглашений (правил), устанавливающих закономерную связь между структурой знака и его денотатом, говорят, что эти соглашения образуют знаковую систему (или язык). При этом правила, определяющие структуру допустимых знаков, называются синтаксисом языка, а правила, определяющие соответствующие допустимым знакам денотаты, называются семантикой языка. (Науку о синтаксисах языков называют синтактикой, а слово «семантика» используется для обозначения как конкретных правил некоторого языка, так и общей науки о таких правилах.)

Одним из примеров знаковой системы служит позиционная система счисления (например, десятичная). Правила, определяющие перечень допустимых цифр и их допустимое расположение (например, справа налево без разделителей), – это

синтаксис. Правила вычисления обозначаемого числа – семантика. При этом запись числа в позиционной системе – знак, а само обозначаемое число – денотат. Известные вам ЯП – также знаковые системы.

Упражнение. Приведите пример синтаксического и семантического правила из таких знаковых систем, как Фортран, Бейсик, Паскаль, Ассемблер.

В общем случае в ЯП знаки – это элементы программ (в том числе полные программы), а денотаты – элементы и свойства поведения исполнителя (атрибуты его поведения), в частности данные, операции, управление, их структура, их связи и атрибуты. Например, знаку, составленному из шести букв «arctan» (элементу программы на Фортране), использованному в этой программе в подходящем контексте, соответствует в качестве денотата такой элемент поведения исполнителя, как вычисление арктангенса.

Знаку, составленному из двух букв «DO» (элементу программы на Фортране), в одном контексте в качестве денотата может соответствовать такой элемент поведения, как вход в цикл, а в другом – переменная вещественного типа, в третьем – массив целого типа.

Упражнение. Выпишите подходящие контексты.

Итак, знаковая система – это правила образования знаков (синтаксис) и согласованные с ними правила образования денотатов (семантика). Подчеркнем, что правила использования денотатов для целей, выходящих за рамки семантики (то есть прагматика), обычно не включаются в знаковую систему. Например, в Фортране нет каких-либо правил, ограничивающих применение соответствующих вычислительных процессов для неблаговидных целей.

Теперь уточненное определение ЯП как знаковой системы для планирования поведения компьютеров должно быть полностью понятным.

1.16. Разновидности программирования

Чтобы создать себе более удобную основу для формирования оценок, принципов и требований, примем соглашения, сужающие область наших рассмотрений.

Во-первых, программировать можно с разной целью. Например, для развлечения и обучения («игровое» программирование, его характерное свойство – интерес не столько к программе-результату, сколько к самому процессу программирования); для отработки идей, приемов, инструментов, методов, критериев, моделей («экспериментальное» программирование, его характерное свойство – созданная программа не предназначена для применения без участия автора, то есть результат такого программирования неотчуждаем). В дальнейшем будем рассматривать только «индустриальное» программирование, цель которого – создание программных изделий (программных продуктов) на заказ или на продажу. Характерное свойство – отчуждаемость результата.

Во-вторых, может быть различным характер использования заготовок программ. По этому критерию различают, по крайней мере, три разновидности программирования:

- **сборочное** – программа составляется из заранее заготовленных модулей (так обычно сейчас работают пакеты прикладных программ);
- **конкретизирующее** – программа получается в результате преобразования универсальных модулей-заготовок (в результате их специализации) в расчете на конкретные условия применения; цель специализации – повышение эффективности (снижение ресурсоемкости) универсальной программы;
- **синтезирующее** – роль заготовок относительно невелика.

В дальнейшем нас, как правило, будет интересовать лишь синтезирующее индустриальное программирование, а также элементы сборочного программирования (когда речь пойдет о модульности).

В-третьих, на различных стадиях жизненного цикла программного изделия (из которого мы выделим стадии проектирования, эксплуатации и сопровождения) предъявляются различные, иногда противоречивые, требования к ЯП. Например, сложность программирования не столь существенна на стадии эксплуатации программы, как ее ресурсоемкость. На стадии проектирования важно удобство написания программ, а на стадии сопровождения – удобство их чтения. В первую очередь будем интересоваться стадией проектирования программного изделия, так как на ней в той или иной форме следует учитывать и требования всех остальных стадий жизненного цикла.

Итак, мы в сущности определили основной стержень нашего интереса, сформулировали основной критерий отбора аспектов ЯП, которым будем уделять в этой книге основное внимание. Нет серьезных оснований претендовать на то, что именно такой стержень обладает какими-то особенными преимуществами. Вдумчивый читатель сможет применить полученные навыки анализа ЯП и при иных исходных соглашениях.

1.17. Понятие о базовом языке

Два выделенных источника сложности – семантический разрыв и незнание мира – полезно трактовать как два различных аспекта единого источника: рассогласования моделей проблемной области (ПО) – области услуг, задач, операций у пользователей и исполнителей.

При таком взгляде создаваемая программа выступает как средство согласования этих моделей. Чем ближе исходные модели, тем проще программа. При идеальном исходном согласовании программа вырождается в прямое указание на одну из заранее заготовленных услуг (например, «распечатать файл», «взять производную», «выдать железнодорожный билет»). Мера рассогласованности моделей положена в основу известной «науки о программах» Холстеда.

Мы уже говорили об исключительном разнообразии моделей даже одного объекта, рассматриваемого с различных точек зрения. Поэтому невозможно построить исполнителя, непосредственно пригодного для выполнения любой услу-

ги. Однако удается строить специализированные исполнители и ориентировать их на фиксированный класс услуг – ПО. Для управления такими специализированными исполнителями создаются проблемно-ориентированные языки (ПОЯ). В качестве хорошо известных примеров укажем язык управления заданиями в операционной системе, язык управления текстовым редактором, язык запросов к базе данных и т. п.

Итак, ПОЯ опирается на определенную модель соответствующей ПО (иногда говорят, что эта модель встроена в такой язык; точнее говоря, ПОЯ – это знаковая система, для которой модель соответствующей ПО служит областью денотатов).

Так что безнадежно строить ЯП с моделями, заготовленными «на все случаи жизни». Однако можно попытаться построить ЯП, на базе которого будет удобно (относительно несложно, с приемлемыми затратами) строить модели весьма разнообразных ПО. Такой язык называют **базовым языком программирования**.

Обычная схема применения базового языка в определенной ПО состоит из двух этапов. На первом (инструментальном) создаются модель ПО и соответствующий ПОЯ (их создают с помощью базового языка программисты-конструкторы). На втором (функциональном) этапе программисты-пользователи решают прикладные задачи, пользуясь созданным ПОЯ. Впрочем, иногда ПОЯ в сущности уже и не язык программирования, так как с его помощью ведут диалог с компьютером, а не планируют поведение заранее. Соответственно, и пользователей такого ПОЯ обычно программистами не называют.

С другой стороны, сам базовый ЯП также можно считать специализированным ПОЯ со своей ПО – он предназначен для построения моделей других ПО и соответствующих ПОЯ. В дальнейшем будем рассматривать именно базовые ЯП, точнее базовые ЯП индустриального программирования.

1.18. Концептуальная схема рассмотрения ЯП

Завершая подготовку к систематическому изучению ряда моделей ЯП, зафиксируем единую схему их рассмотрения. Эта схема поможет сопоставить и оценить различные ЯП прежде всего с точки зрения их пригодности служить базовым языком индустриального программирования.

Если бы нас интересовала, например, оценка ЯП с точки зрения легкости их усвоения начинающими программистами, мы предложили бы, конечно, другую схему их рассмотрения, начав с выявления основных целей и проблем обучения, наиболее подходящих средств решения этих проблем и т. д., подобно нашему пути к базовому языку.

Тем самым предлагаемую ниже схему нужно воспринимать и как демонстрацию существенного элемента систематического метода сравнительной оценки языков. (Конечно, наша учебная схема намного проще той, которую следовало бы строить при практическом решении вопроса о пригодности конкретного ЯП служить базовым языком индустриального программирования.)

Главное назначение базового языка – строить модели ПО, с тем чтобы уменьшить сложность программирования в них. В качестве основных средств понижения сложности мы выделили абстракцию-конкретизацию и прогнозирование-контроль.

Первое средство будем кратко называть аппаратом развития (так как по существу оно служит для построения над исходным языком новой знаковой системы, денотатами в которой выступают введенные абстракции и их конкретизации).

Второе средство будем называть аппаратом защиты (так как оно используется, в частности, для защиты построенных абстракций от разрушения).

Исключительная технологическая роль названных средств дает основание уделить им особое внимание в предлагаемой ниже единой концептуальной схеме рассмотрения ЯП. Опишем начальную версию этой схемы. При необходимости она будет корректироваться и уточняться.

Итак, в каждом ЯП нас будут интересовать три аспекта: **базис**, аппарат развития (просто **развитие**), аппарат защиты (просто **защита**).

Базис ЯП – это предопределенные (встроенные в ЯП) знаки и их денотаты. В базисе будем выделять, во-первых, элементарные типы данных и элементарные операции (это так называемая скалярная сигнатура) и, во-вторых, структуры данных и операций (структурная сигнатура).

Например, в Фортране к скалярной сигнатуре естественно отнести все пять встроенных типов данных, перечень встроенных функций и все операторы. К структурной сигнатуре относятся только массивы и модули. С некоторой натяжкой можно считать компонентой структурной сигнатуры операторы цикла, общие блоки, условные операторы.

Некоторые компоненты базиса составляют аппарат развития ЯП, некоторые – аппарат защиты.

Об аппарате развития уже сказано. Добавим лишь, что будем различать развитие вверх – аппарат определения и использования новых абстракций, и развитие вниз – уточнение и переопределение компонент базиса или ранее определенных абстракций. Например, модуль-подпрограмма в Фортране – средство развития вверх. А средств развития вниз в Фортране нет (в отличие от Ады, Си или CDL).

Об аппарате защиты также уже сказано. Имеется в виду прогнозирование (объявление) свойств поведения объектов (принадлежности к определенному типу, указание области действия, указание ограничений на допустимые значения в определенных контекстах) и контроль за соблюдением ограничений (в частности, управление реакцией на нарушение объявленного поведения).

Кроме базиса, развития и защиты будем рассматривать иногда другие особенности ЯП, в частности особенности эталонного исполнителя ЯП (так называемого языкового процессора) и архитектуру ЯП в целом. Например, для Фортрана исполнитель последовательный, для Рефала – циклический, а для Ады – параллельный.

Архитектуру будем оценивать с точки зрения таких понятий, как цельность (возможность предсказать одни решения авторов языка по другим, согласованность решений), модульность, ортогональность (возможность свободно комбини-

ровать небольшое число относительно независимых фундаментальных выразительных средств) и т. п.

В заключение подчеркнем, что в нашей схеме – только внутренние аспекты ЯП как знаковой системы. Совершенно не затронуты такие важнейшие для выбора и оценки ЯП аспекты, как распространенность на различных типах компьютеров, наличие высококачественных реализаций, уровень их совместимости и т. п.

Глава 2

Пример современного базового ЯП (модель А)

2.1. Общее представление о ЯП Ада 44
2.2. Пример простой программы на Аде 46
2.3. Обзор языка Ада 47
2.4. Пошаговая детализация средствами Ады 52
2.5. Замечания о конструктах 57
2.6. Как пользоваться пакетом управление_сетью 59
2.7. Принцип раздельного определения, реализации и использования услуг (принцип РОРИУС) 66
2.8. Принцип защиты абстракций 67

2.1. Общее представление о ЯП Ада

Наша ближайшая цель – дать как можно более полное представление о современном языке индустриального программирования. Излагаемые принципы и понятия будем демонстрировать на примере ЯП Ада. Он устраивает нас тем, что олицетворяет комплексный подход к решению основной проблемы программирования – проблемы сложности, содержит понятия и конструкты, характерные для интересующего нас класса языков, и к тому же имеет немало шансов стать языком массового программирования в обозримом будущем. Важна для нас и полнота языка в том смысле, что в нем в той или иной форме можно получить ответы на все вопросы современного практического программирования в конкретной ПО. Вместе с тем эти ответы не следует рассматривать как истину в последней инстанции. В других главах мы рассмотрим как критику принятых в Аде решений, так и совершенно иные подходы к программированию в целом.

В соответствии с принципом моделирования ЯП не будем изучать язык Ада полностью, а лишь построим на его основе нашу первую языковую модель – модель А.

Мы исходим из **принципа технологичности** – всякий языковый конструкт предназначен для удовлетворения технологических потребностей на определенных этапах жизненного цикла комплексного программного продукта. Этот принцип, с одной стороны, нацеливает на изучение важнейших потребностей в качестве «заказчиков» понятий и конструктов ЯП. С другой стороны, он требует понимать набор потребностей, обслуживаемых каждым понятием и (или) конструктом. Желательно видеть связь этих понятий с общей идеей абстракции-конкретизации. Будем подчеркивать эту связь, когда посчитаем существенной.

Когда говорят о технологических потребностях, всегда имеют в виду более или менее определенную ПО и технологию решения задач в этой области, в частности технологию программирования. Поэтому следует рассказать об особенностях той ПО, для которой создавался ЯП Ада.

Область, в которой реально применяется ЯП, отнюдь не всегда определяется намерениями его авторов (чаще всего она оказывается пустой!). Однако знание первоначальных замыслов авторов помогает понять особенности ЯП, взглянуть на язык как систему в какой-то мере согласованных решений, почувствовать то, что называют «духом» языка. Иначе язык покажется нагромождением условностей, которые очень трудно запомнить и применять.

Язык Ада создан в основном в 1975–1980 гг. в результате грандиозного проекта, предпринятого МО США с целью разработать единый ЯП для так называемых встроенных систем (то есть систем управления автоматизированными комплексами, работающими в реальном времени). Имелись в виду прежде всего бортовые системы управления военными объектами (кораблями, самолетами, танками, ракетами, снарядами и т. п.). Поэтому решения, принятые авторами Ады, не следует считать универсальными. Их нужно воспринимать в контексте особенностей выбранной ПО.

Характерные требования таковы. Во-первых, необходимы особо надежные и особо эффективные системы управления. Во-вторых, при развитии систем и со-

здании новых очень важно иметь возможность в максимальной степени использовать накопленный программный фонд (проверенные на практике, испытанные программы), то есть программы должны быть относительно легко переносимыми. В-третьих, диапазон сложности систем управления колоссален. В частности, необходимы программы, сложность создания которых требует взаимодействия не только отдельных программистов, но и довольно крупных программистских коллективов. Другими словами, требуется модульность программ – должны быть средства разделения программ на части и комплексации этих частей. В-четвертых, речь идет о системах реального времени – такая система должна управлять поведением реальных физических объектов, движущихся и изменяющихся с весьма высокими скоростями. Таким образом, должна быть предусмотрена возможность вводить параллелизм – рассматривать асинхронные (параллельные) процессы, представляющие такие объекты, и управлять их взаимодействием.

Наконец, в-пятых, в этой области распространена так называемая кросс-технология программирования. Отличается она тем, что программы создаются на одной машине (базовой, инструментальной) для выполнения на другой машине (целевой, объектной). Инструментальная машина обычно обладает существенно более богатыми ресурсами, объектная (часто бортовая, то есть находящаяся непосредственно на управляемом объекте) – более бедными. Это обстоятельство сильно влияет, в частности, на соотношение возможностей периода компиляции и периода выполнения программ (в кросс-технологии компилятор работает на инструментальной машине).

Перечисленные характеристики ПО существенно повлияли на особенности языка Ада. Ада ориентирована на тщательный контроль программ (чтобы повысить надежность встроенных систем), причем такой контроль предусмотрен уже при компиляции (учтены относительно богатые возможности инструментальной машины и бедность объектной). Предусмотрены и возможности оптимизации объектной программы при компиляции (учтены различия инструментальной объектной машины с точки зрения возможностей обеспечить эффективность встроенных систем). Приняты специальные меры, чтобы программы можно было переносить в новые условия эксплуатации при минимальных дополнительных затратах. Имеются средства управления асинхронными процессами. Итак, определяющие требования к языку Ада – надежность, модульность, переносимость, эффективность, параллелизм.

Язык Ада возник в результате международного конкурса языковых проектов, проходившего в 1978–1979 гг. Участники должны были удовлетворить довольно жестким, детально разработанным под эгидой МО США требованиям. Интересно, что все языки, дошедшие до последних туров этого конкурса, были основаны на Паскале. В этой связи Аду можно предварительно охарактеризовать как Паскаль, развитый с учетом перечисленных выше пяти основных требований. При этом авторы (международный коллектив под руководством Жана Ишбиа) пошли в основном по пути расширения Паскаля новыми элементами. В результате получился существенно более сложный язык.

2.2. Пример простой программы на Аде

Чтобы создать у читателя первое зрительное впечатление об Аде, дадим пример совсем простой (но полной) программы. Основой этого (и некоторых других наших примеров) послужили программы учебника Янга «Введение в Аду» [2].

Напишем программу, которая вводит последовательность символов со стандартного устройства ввода и на стандартном устройстве вывода печатает левые и правые скобки, обнаруженные в этой последовательности.

Вот эта программа:

```
1. with  т_обмен;
2. procedure   печать_скобок is
3. ch : символ ;
4. begin
5. т_обмен.дай (ch) ;
6.    while ch /= '.' loop
7.       if ch = '(' or ch = ')' then
8.          т_обмен.возьми (ch) ;
9.       end if ;
10.   т_обмен.дай (ch) ;
11.   end loop  ;
12. end   печать_скобок ;
```

Итак, в программе двенадцать строчек (номера строчек не входят в программу – они нужны, чтобы обсуждать ее свойства). Общий вид программы напоминает текст на Алголе-60 или Паскале (некоторые знакомые ключевые слова, характерная ступенчатая запись и т. п.). Это и неудивительно. Ведь Ада из семейства «поздних алголоидов». Как уже сказано, этот язык – развитие Паскаля, в свою очередь созданного (Никлаусом Виртом) на основе Алгола-60.

Вместе с тем даже в этой простой программе заметны характерные особенности поздних языков индустриального программирования.

Во-первых, высокоразвитая модульность. Фраза с ключевым словом «with» (в переводе с английского «с» или «совместно с») говорит о том, что данную процедуру следует читать, понимать и исполнять во вполне определенном контексте. Этот контекст задан модулем-ПАКЕТОМ с именем «т_обмен». В нем содержатся определения всех ресурсов, необходимых для ввода-вывода текстов (в частности, процедуры «дай» очередной символ со стандартного устройства ввода и «возьми» очередной символ на стандартное устройство вывода). Внутри программы, использующей такой пакет, обращаться к его ресурсам следует по составным именам (сначала название пакета, а затем через точку – название ресурса) как к полям записи в Паскале. При необходимости можно, конечно, ввести сокращенные обозначения для часто используемых ресурсов.

Во-вторых, богатый набор типов данных. В строчке 3 находится объявление переменной ch типа «символ». Это один из предопределенных типов Ады.

Здесь и далее предопределенные идентификаторы языка Ада переведены на русский язык. В оригинале – тип character. Наглядность для нас важнее, чем формальное следование правилам языка; ведь он здесь служит лишь примером общих концепций в ЯП.

Ни в Алголе-60, ни в Фортране такого символьного типа, равноправного с остальными типами, нет. Один из источников выразительной мощи языка Ада – возможность строить новые типы данных, не предопределенные авторами языка. Такая возможность теперь имеется во всех новых ЯП, и мы с ней подробно познакомимся.

В-третьих, ради надежности повышена избыточность, способствующая устранению случайных ошибок. Это и (сколь угодно) длинные названия-идентификаторы, которые можно к тому же составлять из отдельных слов, соединенных одиночным подчеркиванием. Это и строгая скобочная структура текста – каждый управляющий конструкт снабжен специальным «закрывающим» ключевым словом (цикл в строчках с 6 по 11, условный оператор в строчках 7–9, процедура в строчках 2–12). Надежности, ясности, четкой структуре и избыточности способствуют и строгие правила ступенчатой записи программ (в Аде она настоятельно рекомендуется в определении языка и отражена в его синтаксисе).

Смысл программы достаточно очевиден. В строчке 5 вводится первый символ обрабатываемой последовательности и помещается в переменную ch. Далее цикл, работающий до тех пор, пока значением переменной ch не станет символ «.» («/=» – это «не равно», «.» – это признак конца последовательности обрабатываемых символов). В теле цикла – условный оператор, который посылает на устройство вывода очередной символ, если это открывающая или закрывающая скобка. Затем (строкой 10) вводится в переменную ch очередной символ последовательности, и цикл повторяется. Вместе с циклом завершается и процедура печать_скобок.

2.3. Обзор языка Ада

Этот раздел близок по структуре и стилю к разделу 1.4 официального определения языка Ада – национальному стандарту США 1983 г., ставшему в 1986 г. без изменений международным стандартом ИСО. Рассказывая об этом языке и приводя примеры (из различных источников), будем и впредь опираться на это официальное определение. Без существенных изменений оно принято и в качестве отечественного ГОСТа, имеется его перевод на русский язык, отечественные реализации Ады также ориентируются на это определение.

Однако наша цель – не определить язык, а продемонстрировать концепции, характерные (и, как правило, перспективные) для базовых языков индустриального программирования. Поэтому будем стремиться упрощать изложение и избегать деталей, не существенных для нашей задачи.

Само по себе их обилие в официальном сообщении, к сожалению, также характерно. Оно свидетельствует либо о неразвитости науки и практики языкотворчества, либо о фундаментальных свойствах такого социального явления, как ЯП.

Поразительный пример лаконичности – определение Никлаусом Виртом языка Модула-2 [5]).

Итак, нас интересует не столько сам язык Ада, сколько возможность использовать его в качестве источника идей для модели А. Вместе с тем, приводя примеры программ на языке Ада, будем строго следовать стандарту, чтобы не создавать у читателей лишних затруднений при последующем самостоятельном освоении языка. Понятия языка Ада (Ада-понятия) будем выделять прописными буквами.

Ада-программа состоит из одного или более программных МОДУЛЕЙ (сегментов), которые можно компилировать раздельно (кроме задач).

Ада-модуль – это ПОДПРОГРАММА (определяет действия – части отдельных ПРОЦЕССОВ) или ПАКЕТ (определяет часть контекста – совокупность объектов, предназначенных для совместного использования), или ЗАДАЧА (определяет асинхронный процесс), или РОДОВОЙ модуль (заготовка пакета или подпрограммы с параметрами периода компиляции).

В каждом модуле обычно две части: внешность, или СПЕЦИФИКАЦИЯ (содержит сведения, видимые из других модулей), и внутренность, или ТЕЛО (содержит детали реализации, невидимые из других модулей). Разделение спецификации и тела вместе с раздельной компиляцией дает возможность проектировать, писать и проверять программу как набор относительно самостоятельных (слабо зависимых) компонент.

Ада-программа пользуется ТРАНСЛЯЦИОННОЙ БИБЛИОТЕКОЙ. Поэтому в тексте создаваемого модуля следует указывать названия используемых библиотечных модулей.

2.3.1. Модули

Подпрограмма – основной конструкт для определения подпроцессов (ввод данных, обновление значений переменных, вывод данных и т. п.). У подпрограммы могут быть параметры, посредством которых ее связывают с контекстом вызова. Различают две категории подпрограмм – процедуры и функции. Последние отличаются тем, что вырабатывают результат, непосредственно доступный в точке вызова функции. Поэтому вызов функции всегда входит в некоторое выражение.

Пакет – основной конструкт для определения именованного контекста (иногда говорят «логически связанной» совокупности объектов). Несколько расплывчатое «логически связанной» подразумевает возможность объединить в пакет все то, что автор пожелает видеть единым модулем, названным подходящим именем. Это может быть сделано потому, что все связанные в пакет объекты, во-первых, предполагается использовать совместно; во-вторых, необходимо или удобно совместно реализовывать; в-третьих, невозможно или неудобно раздельно определять из-за ограничений на ВИДИМОСТЬ имен. Возможны и иные причины объединения в один пакет определений отдельных имен. Часть из них может быть при этом скрыта, ЗАЩИЩЕНА от непосредственного использования другими модулями; доступ к таким именам строго регламентирован – только через имена, в спецификации пакета явно предназначенные для внешнего использования.

Задача – основной конструкт для определения асинхронного процесса, способного выполняться параллельно с другими процессами. Процессом называется определенным образом идентифицируемая последовательность действий исполнителя, линейно-упорядоченная во времени. В одном модуле-задаче можно определить один асинхронный процесс или совокупность аналогичных асинхронных процессов (так называемый ЗАДАЧНЫЙ ТИП). **Асинхронность можно обеспечивать как отдельными процессорами для каждого процесса, так и «прерывистым» выполнением различных процессов на одном процессоре.**

2.3.2. Объявления и операторы

В теле модуля в общем случае две части – ОБЪЯВЛЕНИЯ и ОПЕРАТОРЫ.

Объявления вводят новые знаки (ИМЕНА) и связывают их с денотатами (ОБЪЕКТАМИ). Эта связь имени с определенным объектом (знаковая ситуация) сохраняет силу в пределах ОБЛАСТИ ДЕЙСТВИЯ имени. Таким образом, формально объект – это то, что можно именовать. Вместе с тем авторы языка Ада стремились к тому, чтобы ада-объектами было удобно представлять содержательные объекты решаемой задачи. Ада-объектами могут быть, в частности, ПОСТОЯННАЯ, ПЕРЕМЕННАЯ, ТИП, ИСКЛЮЧЕНИЕ, ПОДПРОГРАММА, ПАКЕТ, ЗАДАЧА и РОДОВОЙ модуль.

Операторы предписывают действия, которые выполняются в порядке следования операторов в тексте программы (если только операторы ВЫХОДА из конструкта (exit), ВОЗВРАТА (return), ПЕРЕХОДА по метке (go to) или возникновения исключения (исключительной ситуации) не заставят продолжить исполнение с другого места).

Оператор ПРИСВАИВАНИЯ изменяет значение переменной.

ВЫЗОВ ПРОЦЕДУРЫ активизирует исполнение соответствующей процедуры после связывания каждого фактического параметра (АРГУМЕНТА) с соответствующим формальным параметром (ПАРАМЕТРОМ).

УСЛОВНЫЙ (if) и ВЫБИРАЮЩИЙ (case) операторы позволяют выбрать одну из возможных вложенных последовательностей операторов в зависимости от значения УПРАВЛЯЮЩЕГО ВЫРАЖЕНИЯ (условия).

Оператор ЦИКЛА – основной конструкт для описания повторяющихся действий. Он предписывает повторять указанные в его теле действия до тех пор, пока не будет выполнен оператор выхода, явно указанный в теле цикла, или не станет истинным условие окончания цикла.

Блочный оператор (БЛОК) соединяет последовательность операторов с непосредственно предшествующими ей объявлениями в единую ОБЛАСТЬ ЛОКАЛИЗАЦИИ. Объявленные в ней объекты считаются ЛОКАЛЬНЫМИ в этой области.

Имеются операторы, обслуживающие взаимодействие асинхронных процессов.

При исполнении модуля могут возникать ошибочные ситуации, в которых нельзя нормально продолжать работу. Например, возможно арифметическое переполнение или попытка получить доступ к компоненте массива с несуществую-

щим индексом. Для обработки таких ИСКЛЮЧЕНИЙ (исключительных ситуаций) в конце сегментов можно разместить специальные операторы РЕАКЦИИ на исключение (exception). Имеются и явные операторы ВОЗБУЖДЕНИЯ исключений (raise). Они включают в действие аппарат обработки возбужденного исключения.

2.3.3. Типы данных

Среди объектов языка Ада можно выделить ОБЪЕКТЫ ДАННЫХ (то есть объекты, которым разрешено играть роль данных по отношению к каким-либо операциям). Каждый объект данных в Аде характеризуется определенным ТИПОМ. Своеобразие этого языка в значительной степени связано именно с системой типов. Для тех, кто работал только с Фортраном, Алголом и Бейсиком, многое в этой системе окажется совершенно незнакомым. В частности, возможность определять новые типы, отражающие особенности решаемой задачи. Для освоивших Паскаль адовские типы привычнее, но система адовских типов полнее и строже.

Тип, с одной стороны, – важнейшая компонента аппарата прогнозирования-контроля. Приписывая объекту данных определенный тип, ограничивают его возможное поведение. С другой стороны, зная тип, получают возможность это поведение контролировать. Наконец, зная ограничения на возможное поведение, можно рационально выделять память и другие ресурсы. С типом в Аде связывают три основных ограничения.

Тип ограничивает, во-первых, ОБЛАСТЬ ЗНАЧЕНИЙ объекта; во-вторых, НАБОР ОПЕРАЦИЙ, в которых объекту разрешено фигурировать; в-третьих, набор допустимых для него ролей в этих операциях (второй операнд, результат и т. п.).

Имеются четыре категории типов: СКАЛЯРНЫЕ (в том числе ПЕРЕЧИСЛЯЕМЫЕ и ЧИСЛОВЫЕ), СОСТАВНЫЕ (в том числе РЕГУЛЯРНЫЕ (массивы) и КОМБИНИРОВАННЫЕ (записи, структуры)), ССЫЛОЧНЫЕ (указатели) и ПРИВАТНЫЕ (закрытые, защищенные – их представление для пользователя невидимо).

Скалярные типы. Когда определяют перечисляемый тип, явно указывают перечень лексем, которые и составляют область возможных значений объектов вводимого типа. Такой перечень может быть списком дней недели (пн, вт, ср, чт, пт, сб, вс), списком символов некоторого алфавита ('A','B',...,'Z') и т. п. Перечисляемые типы избавляют программиста от необходимости кодировать содержательные объекты целыми числами. Перечисляемые типы BOOLEAN (логический) и CHARACTER (символьный) считаются ПРЕДОПРЕДЕЛЕННЫМИ, то есть встроенными в язык и действующими без предварительного явного объявления в программе. Набор символов типа CHARACTER соответствует алфавиту ASCII – Американскому стандартному коду для обмена информацией.

Числовые типы обеспечивают точные и приближенные вычисления. В точных вычислениях пользуются ЦЕЛЫМИ типами. Область возможных значений для

таких типов – конечный диапазон целых чисел. В приближенных вычислениях пользуются либо АБСОЛЮТНЫМИ типами (задается абсолютная допустимая погрешность), либо ОТНОСИТЕЛЬНЫМИ типами (задается относительная погрешность). Абсолютная погрешность задается явно и называется ДЕЛЬТОЙ, относительная погрешность вычисляется по заданному допустимому количеству значащих цифр в представлении числа. Подразумевается, что абсолютные типы будут представлены машинной арифметикой с фиксированной точкой, а относительные – с плавающей. Числовые типы INTEGER (целый), FLOAT (плавающий) и DURATION (временные задержки для управления задачами) считаются предопределенными.

Составные типы. Скалярные типы (и перечисляемые, и числовые) выделяются тем, что объекты этих типов считаются атомарными (не имеющими составляющих). Составные типы, в отличие от скалярных, позволяют определять структурированные объекты (массивы и записи). Массивы служат значениями регулярных типов – компоненты массивов доступны по индексам. «Регулярность» массивов проявляется в том, что все компоненты должны быть одного типа. Записи (структуры) служат значениями комбинированных типов – их компоненты могут быть различных типов; компоненты записей доступны по именам-селекторам. Имена компонент одной и той же записи должны быть различны; компоненты называются также ПОЛЯМИ записи.

Строение записей одного типа может зависеть от значений выделенных полей, называемых ДИСКРИМИНАНТАМИ. Дискриминанты играют роль параметров комбинированного типа – задавая набор дискриминантов, выбирают определенный вариант структуры объектов этого типа. Поэтому типы с дискриминантами называют также ВАРИАНТНЫМИ типами.

Ссылочные типы. Если структура объектов составных типов (в случае вариантных типов – все варианты такой структуры) фиксируется статически (то есть до начала выполнения программы), то ссылочные типы позволяют создавать и связывать объекты динамически (при исполнении программы, точнее при исполнении ГЕНЕРАТОРОВ). Тем самым появляется возможность динамически создавать сколь угодно сложные конгломераты объектов. Генератор создает объект указанного (статически известного) типа и обеспечивает доступ к вновь созданному объекту через переменную соответствующего ссылочного типа. Передавая (присваивая) ссылки, можно организовывать произвольные структуры. Важно, что и элементы, и связи в таких динамических структурах можно менять при исполнении программы.

Приватные типы. Доступ к ПРИВАТНЫМ объектам (их называют также абстрактными объектами, а соответствующие типы – абстрактными типами данных (АТД)) находится под полным контролем автора приватного типа. Такой тип всегда определяется в пакете, который называется ОПРЕДЕЛЯЮЩИМ пакетом для этого типа.

Спецификация определяющего пакета фиксирует полный набор операций и тех ролей в этих операциях, в которых могут фигурировать объекты нового типа. В определяющем пакете фиксируется и реализация приватного типа, однако в ис-

пользующих этот пакет модулях она непосредственно недоступна – только через явно перечисленные автором пакета допустимые операции. Поэтому реализацию можно изменять, не заставляя переделывать использующие модули.

Концепция типа в Аде дополнена аппаратом ПОДТИПОВ (они ограничивают область значений, не затрагивая допустимых операций), а также аппаратом ПРОИЗВОДНЫХ типов (они образуются из уже известных типов, наследуя связанные с ними значения и операции).

Поговорим об остальных средствах языка. Посредством УКАЗАТЕЛЯ ПРЕДСТАВЛЕНИЯ можно уточнить требования к реализации определенных типов на целевой машине. Например, можно указать, что объекты такого-то типа следует представить заданным количеством битов, что такие-то поля записи должны располагаться с такого-то адреса. Можно указать и другие детали реализации, вплоть до прямой вставки машинных команд. Ясно, что, с одной стороны, подробное описание представления мешает переносу программы в другую операционную обстановку. С другой – оно может оказаться исключительно важным для качества и даже работоспособности программы. Явное указание представления помогает отделять машинно независимые части модулей от машинно зависимых. В идеале только указатели представления и потребуется менять при переносе программы.

Обмен с внешней средой (ввод-вывод) обслуживается в Аде предопределенными библиотечными пакетами. Имеются средства обмена значениями не только предопределенных типов (как в Паскале), но и типов, определяемых программистом.

Наконец, имеются средства статической параметризации модулей (действующие до начала исполнения программы, в период компиляции) – аппарат РОДОВЫХ модулей. Параметры таких модулей (родовые параметры), в отличие от динамических параметров подпрограмм и процедур, могут быть не только объектами данных некоторого типа, но и такими объектами, как типы и подпрограммы (которые в Аде не считаются объектами данных). Так что общие модули, рассчитанные на применение ко всем типам данных определенной категории, в Аде следует оформлять как родовые.

На этом закончим краткий обзор языка.

2.4. Пошаговая детализация средствами Ады

Рассмотрим следующую задачу.

Содержательная постановка. Необходимо предоставить пользователю комплекс услуг, позволяющих ему моделировать сеть связи. Пользователь должен иметь возможность изменять сеть (добавлять и убирать узлы и линии связи), а также получать информацию о текущем состоянии сети.

Требования к реализации. Во-первых, должна быть гарантирована целостность сети при любых возможных действиях пользователя. Другими словами, ни при каких условиях не должны возникать линии связи, не ведущие ни в один узел.

Во-вторых, предполагается развивать возможности моделирования (например, отражать стоимость связей). Важно, чтобы при этом у пользователя не возникала необходимость изменять готовые программы.

Оба этих требования можно удовлетворить, если строго регламентировать доступ пользователя к представлению сети в памяти компьютера. Тогда заботу о целостности сети можно возложить на средства доступа к ней, а при развитии комплекса услуг можно изменять представление сети, сохраняя все старые средства доступа (и, следовательно, ранее работавшие программы пользователя).

Первый шаг детализации. Уточним постановку задачи в терминах языка Ада. Так как речь идет не об алгоритме, а о предоставлении пользователю комплекса услуг, в Ада-терминах естественно отобразить этот комплекс на совокупность «логически связанных» объектов, в данном случае – связанных по меньшей мере совместным использованием. Другими словами, первое наше решение – создавать ПАКЕТ, а не подпрограмму или задачу. Вспоминая, что разделение спецификации и тела пакета позволит скрыть от пользователей пакета детали реализации (в частности, представление сети, в полном соответствии с требованиями), получаем еще одно подтверждение, что решение правильное.

Итак, создаем ПАКЕТ. Нужно придумать ему название, выражающее назначение предоставляемого комплекса услуг. Попробуем «сеть». Нехорошо. По-видимому, так лучше называть тот объект, который будет моделироваться и чье представление нужно скрыть в теле нашего пакета. Попробуем «моделирование сети связи». Лучше, но слишком конкретно. Хотя в постановке задачи и требованиях речь идет именно о моделировании сети связи, однако специфика связи (кроме самой сети) ни в чем не отражена (нет и речи о пропускной способности каналов, классификации сообщений и т. п.), да и специфика моделирования не затронута (никаких моделей отправителей, получателей и т. п.). Скорее, мы собираемся предоставить лишь комплекс услуг по управлению сетью. Так и назовем пакет:

«управление_сетью».

Точное название настраивает на то, чтобы в пакете не было лишнего, а пользователю помогает применять наш комплекс и в других областях.

Второй шаг детализации. Теперь нужно написать спецификацию пакета, объявив все объекты, с которыми сможет работать пользователь:

```
0.     with параметры_сети; use параметры_сети;
1.     package управление_сетью is
2.     type узел is new INTEGER range 1..макс_узлов;
3.     type число_связей is new INTEGER range 0..макс_связей;
4.     type индекс_узла is new INTEGER range 1..макс_связей;
5.     type перечень_связей is array (индекс_узла) of узел;
6.     type связи is
7.        record
8.           число : число_связей := 0;
9.           узлы : перечень_связей;
10.       end record ;
```

```
11.-- операции над сетью
12.   procedure вставить (X : in узел);
13.   procedure удалить (X : in узел);
14.   procedure связать (X, Y : in узел);
15.-- сведения о текущем состоянии сети
16.   function узел_есть (X : узел) return boolean;
17.   function все_связи (X : узел) return связи;
18. end управление_сетью;
```

Текст спецификации пакета с названием «управление_сетью» при первоначальном знакомстве может показаться непонятным. Во всяком случае, не верится, что он получен одним шагом детализации. Действительно, шаг зависит как от привычки проектировщика, так и от свойств языка. Ведь в общем случае далеко не каждый мелкий шаг поддерживается подходящим законченным языковым конструктом. Например, в Аде шаги нельзя дробить сколь угодно мелко хотя бы потому, что действует **правило последовательного определения**: при очередном определении можно использовать только предопределенные, внешние или ранее объявленные имена.

Но при пошаговой детализации нельзя заранее объявить те имена, которые понадобятся, – они попросту неизвестны. Когда проектируют совокупность модулей, это не помеха (порядок модулей несуществен). А вот внутри модулей правило последовательного определения мешает пошаговой детализации (особенно внутри пакетов; **почему?**). Приходится либо применять средства, выходящие за рамки Ады (например, псевдокод), либо записывать пакет «с конца к началу» – этот порядок с учетом правила последовательного определения лучше отражает последовательность появления имен при пошаговой детализации.

> С точки зрения принципа технологичности, любые несоответствия языка потребностям пошаговой детализации служат источником «точек роста», намечают направление развития либо самого языка, либо других связанных с ним технологических инструментов. Для Ады, в частности, разрабатываются специальные средства поддержки пошагового конструирования программ.
>
> **Упражнение.** Укажите внешний эффект (исходные данные и результаты) хотя бы одного из таких средств.
>
> **Подсказка.** Редактор, располагающий фрагменты Ада-программы в порядке, соответствующем правилу последовательного определения.
>
> **Упражнение (повышенной сложности).** Разработайте проект хотя бы одного такого средства; проект комплекса таких средств.

Итак, проявим более мелкие шаги проектирования нашего пакета.

Шаг 2.1 (строка 17). Объявляем функцию с названием «все_связи», формальным параметром с названием «X» (значениям этого параметра приписан тип с названием «узел») и результатом, которому приписан тип с названием «связи».

> Ниже будем писать короче: функцию «все_связи» с параметром «X» типа «узел» и результатом типа «связи».

Эта функция дает возможность узнавать о текущем состоянии сети (точнее, о связях одного узла). Обратите внимание, пока совершенно не ясно, что такое «узел» и «связи». Это лишь названия, отражающие роль в создаваемом комплексе услуг тех объектов, которые еще предстоит воплотить в программе.

Шаг 2.2 (строка 16). Нехорошо запрашивать связи узла, не зная, имеется ли он в сети. Поэтому (продолжая предоставлять средства узнавать о состоянии сети) объявляем функцию узел_есть с параметром «Х» типа узел и результатом логического типа (BOOLEAN).

Замечание. Обратите внимание, мы записываем только формальные СПЕЦИФИКАЦИИ (заголовки) функций. Содержащихся в них сведений достаточно, чтобы можно было (столь же формально) написать вызов такой функции. Но, во-первых, рано или поздно придется написать ТЕЛО функции (сделаем это в ТЕЛЕ пакета). Во-вторых, нужно как-то сообщить пользователю, что же делает объявляемая функция.
Например, из объявления в строке 16 пользователь поймет, что, задав функции узел_есть аргумент типа узел, он получит в качестве результата истину или ложь. Неоткуда ему узнать, что истина соответствует случаю, когда узел с указанным именем есть в сети, а ложь – когда его нет. Название функции лишь намекает на такое истолкование. Конечно, названия должны быть мнемоничными и помогать запоминать смысл программных объектов, но они не могут заменить точных сведений.

Ада не предоставляет специальных средств для полного и точного описания внешнего эффекта модуля. Ведь адовские спецификации рассчитаны прежде всего на исполнителя, на компьютер, а отнюдь не на пользователя. Поэтому, как и в случае с другими ЯП, проектирование Ада-модуля следует сопровождать проектированием точного описания его внешнего эффекта (применяя при необходимости средства, выходящие за рамки ЯП). Некоторые экспериментальные языки предоставляют встроенные средства соответствующего назначения.

Часто бывает необходимо параллельно создавать и описание внешнего эффекта, специально ориентированное на пользователей. Эта так называемая пользовательская документация принципиально отличается от описаний, рассчитанных на автомат-транслятор (именно таковы описания на ЯП) или человека-реализатора, по структуре документов, стилю изложения, выделяемым свойствам объектов и т. п.

С точки зрения пользовательской документации на программное изделие, ЯП всегда выступает в роли инструмента реализации. Он тем лучше, чем проще объяснить пользователю назначение выделенных программных компонент и чем ими удобнее и дешевле пользоваться. Назовем соответствующий критерий качества языка *критерием выделимости*.

По выделимости Ада превосходит, например, Алгол-60 или Бейсик, так как позволяет адекватно оформлять не только компоненты-функции, но и компоненты-данные, и компоненты-задачи, и компоненты более «тонкого» назначения. Другими словами, Ада выигрывает в выделимости потому, что предоставляет более развитые средства абстракции и конкретизации.

Упражнение. При чем здесь абстракция-конкретизация?

Шаг 2.3 (строки 12–14). Предоставляя средства для изменения сети, определяем три процедуры: вставить, удалить и связать (параметры у них типа узел).

С одной стороны, после этого шага мы можем быть довольны – внешние требования к проектируемому комплексу услуг в первом приближении выполнены. С другой стороны, появилась необходимость определить упомянутые на предыдущих шагах типы данных.

> Так всегда – завершая абстракцию и конкретизацию верхнего уровня, создаем почву для аналогичной работы на нижнем уровне, и наоборот.

Шаг 2.4 (строка 2). Определяем тип «узел». Этот тип уже частично нами охарактеризован **(где?)** – данные типа «узел» могут служить аргументами всех процедур и функций, объявленных в нашем пакете. Другими словами, рассматриваемый тип уже охарактеризован по фактору применимых операций. Выписывая его явное определение, мы характеризуем данные этого типа по фактору изменчивости – указываем, что диапазон (range) их возможных значений – целые числа от 1 до числа макс_узлов (пока еще не определенного). Одновременно мы относим объявляемый тип к категории целых числовых типов и тем самым завершаем его характеристику по фактору применимых операций (в Аде для целых типов предопределены обычные операции целой арифметики – сложение «+», вычитание «–», умножение «*» и др.).

Шаг 2.5 (строки 6–10). Определяем тип «связи» результата функции все_связи. Замысел в том, чтобы эта функция сообщала число связей указанного узла и перечень связанных с ним узлов. В Алголе-60 или Фортране не могло быть функций, которые в качестве результата выдают составной объект. В Аде можно ввести составной тип, объекты которого состоят либо из однотипных подобъектов – являются массивами, либо из разнотипных – являются записями. Результат задуманной нами функции все_связи – пара разнотипных объектов (число и узлы). Другими словами, это запись, первое поле которой называется «число», а второе – «узлы». Тип значений первого поля назван «число_связей», второго – «перечень_связей».

В этом же объявлении указано, что при создании объекта типа «связи» его поле «число» получает начальное значение 0. Это так называемая ИНИЦИАЛИЗАЦИЯ объектов, которой нет, например, в Алголе-60, но для знающих Фортран – дело привычное (вспомните объявление начальных данных DATA).

Итак, на шаге 2.5 снова кое-что определилось, но опять появились новые имена – число_связей и перечень_связей.

Шаг 2.6 (строка 5). Перечень_связей определяем как регулярный тип одномерных массивов, составленных из объектов типа узел, доступ к которым – по индексам типа индекс_узла.

Шаг 2.7 (строка 4). Индекс_узла определяем как тип объектов, значения которых лежат в диапазоне целых чисел от 1 до макс_связей (максимального числа связей у узла в сети – оно пока не определено).

Шаг 2.8 (строка 3). Число_связей определяем как тип объектов, значения которых лежат в диапазоне целых чисел от 0 до макс_связей. Как видите, этот тип похож на предыдущий, но отличается своей ролью и диапазоном значений.

Остались неопределенными только имена макс_узлов и макс_связей. Их неудобно фиксировать в том же модуле – ведь они могут изменяться в зависимости от потребностей пользователя и наличных ресурсов. Поэтому будем считать, что эти имена определены во внешнем для нашего модуля контексте, а именно в пакете с именем «параметры_сети». Доступ к этому контексту из модуля управление_сетью обеспечивается его нулевой строкой.

Это так называемое УКАЗАНИЕ КОНТЕКСТА. После ключевого слова with в нем перечисляются пакеты, объявления из которых считаются видимыми в модуле, непосредственно следующем за таким указанием.

Пакет параметры_сети можно определить, например, так:

```
1. package параметры_сети is
2.    макс_узлов : constant INTEGER := 100;
3.    макс_связей: constant INTEGER := 8;
4. end параметры_сети;
```

Тем самым макс_узлов определено в качестве ПОСТОЯННОЙ целого типа со значением 100, а макс_связей – в качестве постоянной того же типа со значением 8. Значения постоянных нельзя менять при исполнении программы (вот еще один элемент прогнозирования и контроля в Аде).

2.5. Замечания о конструктах

Рассмотрим написанные фрагменты программы еще раз. Теперь поговорим о строении, смысле и назначении использованных конструктов.

В целом мы написали две СПЕЦИФИКАЦИИ ПАКЕТА. Отличительный признак этого конструкта – ключевое слово package (пакет). Спецификация пакета содержит объявления имен, которые становятся доступными при использовании пакета посредством указания контекста (например, объявления из пакета параметры_сети становятся доступны в пакете управление_сетью, если указать контекст with параметры_сети).

Спецификацию пакета можно оттранслировать и поместить в ТРАНСЛЯЦИОННУЮ БИБЛИОТЕКУ. Получится модуль, пригодный для связывания (посредством указаний контекста) с другими (использующими его) модулями в процессе их трансляции и загрузки.

Пакет может состоять из одной спецификации или из спецификации и тела. Например, для пакета параметры_сети тело не требуется, в отличие от пакета управление_сетью (**как вы думаете, почему?**).

Если пакет состоит из двух частей (спецификации и тела), то выполнять программу, в которой отсутствует одна из них, нельзя. Однако для трансляции использующих модулей достаточно одной только спецификации используемого пакета. Итак, создавать и транслировать спецификации пакетов можно отдельно от их тел, но исполнять – только совместно с телами пакетов. В спецификацию пакета входит совокупность ОБЪЯВЛЕНИЙ.

Так, в каждой из строк 2–3 спецификации пакета параметры_сети находится ОБЪЯВЛЕНИЕ ПОСТОЯННОЙ, точнее ОБЪЯВЛЕНИЕ ЧИСЛА. Это одна из

разновидностей ОБЪЯВЛЕНИЯ ОБЪЕКТА. Назначение всякого объявления объекта – связать имя с характеристиками поведения объекта, названного этим именем. Поэтому обязательными компонентами объявления служат само вводимое имя, ключевые слова, отличающие разновидность объявления и тем самым характеризующие поведение объявляемого объекта в целом, и компоненты-параметры, уточняющие характеристики поведения.

Так, в строке 2 объявляемое имя – макс_узлов, уточняющие параметры – имя типа (INTEGER) и константа 100 (изображение целого числа). Полное объявление связывает с именем объекта макс_узлов тип INTEGER и константу 100 как характеристику поведения объекта. Попросту говоря, имя макс_узлов начинает обозначать константу 100 типа INTEGER.

> Чтобы понять, зачем нужно обозначать константы именами, достаточно представить себе программу, где константа 100 используется в десяти местах, и допустить, что нужно изменить ее значение на 200. Тогда в нашей спецификации достаточно изменить одну цифру в строке 2, а иначе пришлось бы изменять десять мест с риском где-нибудь заменить не ту константу (или не на то значение). Так объявления постоянных способствуют надежности Ада-программ.

Вернемся к спецификации пакета управление_сетью (на стр. 53). В каждой из ее строк 2, 3 и 4 мы написали ОБЪЯВЛЕНИЕ ТИПА. В нем всегда указывают, как совокупность значений объявляемого типа образуется из совокупности значений ранее известных типов (предопределенных или ранее объявленных). В нашем случае в строке 2 указано, что новый тип «узел» образован из предопределенного типа INTEGER (является типом, ПРОИЗВОДНЫМ от типа INTEGER), причем данные типа «узел» могут обозначать только целые из диапазона от 1 до макс_узлов. В строках 3 и 4 аналогичные сведения сообщаются о типах число_связей и индекс_узла, только здесь указаны другие диапазоны.

Напомним, зачем нужны объявления типов. В том модуле, где будет использоваться пакет управление_сетью, можно объявить переменную (например, А) типа «узел» и переменную (например, В) типа число_связей. Так вот переменную А можно указать в качестве аргумента процедуры «вставить» или «связать», а переменную В – нельзя. Это ошибка, обнаруживаемая при трансляции. В сущности, ради такого контроля и нужны объявления типов, прогнозирующие поведение (возможные роли) соответствующих данных.

В строке 5 – объявление типа, но на этот раз не скалярного (как в строках 2–4), а СОСТАВНОГО, точнее РЕГУЛЯРНОГО. Указано, как значения нового типа перечень_связей образуются из значений типов «узел» и индекс_узла. Именно значения типа перечень_связей – это одномерные (так как указан лишь один диапазон индексов) МАССИВЫ, компонентами которых служат значения типа узел, а доступ к этим компонентам – по индексам типа индекс_узла.

В строках 6–10 – также объявление составного типа, но на этот раз – КОМБИНИРОВАННОГО. Указано, что значениями нового типа «связи» могут быть любые ЗАПИСИ с двумя полями. Первое поле с именем «число» и допустимыми значениями типа «число_связей» (при создании записи этому полю присваивается начальное значение 0). Второе поле с именем «узлы» типа перечень_связей.

Если в модуле, использующем наш пакет, объявлена переменная, например X типа «связи» и I типа индекс_узла, то через X.узлы(I) обозначается значение типа узел, которое служит I компонентой поля «узлы» переменной X.

Строки 11 и 15 – это примечания, не влияющие на смысл модуля. Примечанием считается остаток любой строки, начинающийся с двух минусов.

> Иногда мы будем переносить примечания на следующие строчки, не предваряя продолжение примечаний двумя дефисами. Стандарт Ады такого не допускает.

В строках 12–14 – ОБЪЯВЛЕНИЯ ПРОЦЕДУР. В скобках указаны имена (названия) формальных параметров, их типы и РЕЖИМ использования (in – только для чтения – ВХОДНЫЕ параметры; out – только для записи – ВЫХОДНЫЕ; in out – и для чтения, и для записи – ОБНОВЛЯЕМЫЕ). Режим in напоминает вызов параметров значением в Алголе или Паскале, in out – вызов параметров со спецификацией var в Паскале или ссылкой в Фортране, out – точного аналога в этих ЯП не имеет.

В строках 16–17 – ОБЪЯВЛЕНИЯ ФУНКЦИЙ. Отличаются от процедур ключевым словом function (а не procedure) и указанием типа результата (после return). Режим параметров не указывается, потому что для функций всегда подразумевается режим in (все параметры функций – только входные, то есть функции не могут менять значения своих аргументов).

Обратите внимание, в спецификации пакета указаны лишь спецификации (заголовки) процедур и функций. В таком случае их тела следует поместить в ТЕЛО ПАКЕТА, о котором пойдет речь в следующем разделе.

На этом закончим предварительное знакомство с Ада-конструктами.

2.6. Как пользоваться пакетом управление_сетью

Пусть нужно построить сеть из пяти узлов (13, 33, 25, 50, 90) и шести дуг (13, 33), (33, 25), (33, 50), (33, 90), (13, 50) и (25, 90). **(Нарисуйте такую сеть.)**

Это можно сделать следующей процедурой построение_сети:

```
with управление_сетью;
use управление_сетью;
procedure построение_сети is
begin
   вставить(33); вставить(13);
   связать(33,13); вставить(25);
   связать(33,25); вставить(50);
   вставить(90); связать(33,50);
   связать(33,90); связать(13,50);
   связать(25,90);
end построение_сети;
```

Первые две строки позволяют пользоваться услугами, предоставляемыми пакетом управление_сетью, так, как будто все услуги объявлены непосредственно перед третьей строкой.

Как уже сказано, строка с ключевым словом with называется УКАЗАНИЕМ КОНТЕКСТА (with clause). Указание контекста делает видимыми (доступными по ПОЛНЫМ именам) все услуги, объявленные в пакетах, перечисленных вслед за with. Например, к процедуре «вставить» можно было бы обратиться так:

```
управление_сетью.вставить(...);
```

а объявить переменную типа «связи» можно так:

```
A : управление_ сетью.связи;
```

Строка с ключевым словом use называется УКАЗАНИЕМ СОКРАЩЕНИЙ (use clause). Это указание позволяет пользоваться видимыми именами, не предваряя их именем пакета. Так мы и поступили в процедуре построение_сети. Подчеркнем, что указание сокращений действует только для уже видимых имен. Его обязательно нужно предварять указанием контекста.

Если нужно напечатать сведения о построенной сети, то это можно сделать следующими операторами (будем считать, что предопределены процедуры новая_строка (переход на новую строку при печати) и «печать» (целого числа или массива)):

```
новая_строка;
for i in узел loop
   if узел_есть(i) then
      печать(i);
      печать(все_связи(i).узлы);
   end if;
   новая_строка;
end loop;
```

Будет напечатано:

```
13 33 50
25 33 90
33 13 25 50 90
50 33 13
90 33 25
```

Обратите внимание, тип «узел» используется для указания диапазона изменения значений переменной цикла. В нашем случае тело цикла выполнится 100 раз.

Третий шаг детализации – тело пакета. До сих пор мы смотрели на наш комплекс услуг с точки зрения потенциального пользователя. Теперь настало время реализовать те услуги, которые мы объявили в спецификации пакета. В терминах Ады это означает, что нужно спроектировать ТЕЛО ПАКЕТА управление_сетью. Создавать тело пакета будем также пошаговой детализацией.

Шаг 3.1. Не важно, с детализации какой процедуры или функции начинать, – ведь ни одну из них нельзя написать прежде, чем не станет понятно, как представлена сама сеть, с которой нужно работать. Поэтому начнем с проектирования представления данных. Займемся представлением сети.

Есть много вариантов такого представления (таблица, список, перемешанная таблица и т. п.). Выберем представление сети массивом:

```
сеть : array (узел) of запись_об_узле;
```

Мы написали ОБЪЯВЛЕНИЕ ОБЪЕКТА. Как всякое объявление объекта, оно связывает имя («сеть») с характеристиками того объекта данных, который в дальнейшем будет значением (денотатом) объявленного имени. В нашем случае этот объект – одномерный массив с компонентами типа запись_об_узле, доступными по индексам типа узел.

Шаг 3.2. Следует заняться уточнением того, как устроен объект типа запись_об_узле. Естественно считать, что это некоторая структура данных, куда вносятся сведения о том, включен ли узел в сеть, если да, то какие узлы с ним связаны. Объявим тип запись_об_узле.

```
type запись_об_узле is record
   включен : BOOLEAN := false;
   связан  : связи;
end record;
```

Итак, каждая запись об узле состоит из двух полей. Поле с именем «включен» и начальным значением false служит признаком включения узла в сеть, а поле с именем «связан» содержит все связи узла.

Шаг 3.3. Теперь все готово, чтобы заняться операциями над сетью. Начнем с функции узел_есть.

Уточним ее внешний эффект: она должна быть применима к любому объекту типа «узел» и должна выдавать результат true, если узел с таким именем есть в сети, и false в противном случае.

Мы сформулировали ее содержательный эффект. Такого рода сведения о функции узел_есть должны быть в пользовательской документации. Это необходимое для пользователя дополнение к спецификации (заголовку функции), указанной в спецификации пакета в строке 18. Но сейчас нас интересует реализация функции. Поэтому следует обеспечить ее содержательный эффект в реализационных терминах, в частности через представление сети (которое пользователю недоступно и даже может оказаться неизвестным). Было бы естественным выдавать в качестве результата просто значение поля «включен» записи об узле. Но для этого на всю остальную реализацию пакета необходимо наложить единое требование (если угодно, определить дисциплину работы с этим полем): его значением в любой компоненте массива «сеть» после выполнения любого действия должно быть true, если узел есть в сети, и false в противном случае. При выполнении этого требования необходимый содержательный внешний эффект функции узел_есть обеспечивается следующим объявлением (определением):

```
function узел_есть(X : узел) return BOOLEAN is
begin
   return сеть(X).включен;
end узел_есть;
```

Обратите внимание, в полном определении функции повторена ее спецификация.

ОПЕРАТОР ВОЗВРАТА (return) завершает исполнение тела функции, доставляя в качестве ее результата значение указанного выражения. В нашем случае

это ВЫБОРКА (поля «включен» из записи, находящейся в массиве «сеть» по индексу, указываемому значением формального параметра «X»).

Шаг 3.4. Займемся реализацией функции все_связи. Ее содержательный внешний эффект – проявление связей узла. При соответствующей дисциплине работы с сетью ее реализация могла бы быть такой:

```
function все_связи(X : узел) return связи is
begin
   return сеть(X).связан;
end все_связи;
```

Вопрос. В чем должна состоять требуемая дисциплина?

К такой функции можно обращаться лишь тогда, когда известно, что узел в сети есть, иначе можно выбрать неопределенное значение в поле «связан».

Шаг 3.5. Реализация процедуры «вставить» (с очевидным содержательным эффектом) может выглядеть так:

```
procedure вставить(X : in узел) is
begin
   сеть(X).включен := true;
   сеть(X).связан.число := 0;
end вставить;
```

Теперь займемся процедурами «удалить» и «связать». Они чуть сложнее за счет того, что нужно вносить изменения в несколько компонент массива «сеть».

Шаг 3.6. Содержательный эффект процедуры «удалить» очевиден: узел с указанным именем должен быть удален из сети и (с учетом требования поддерживать целостность сети) все связи, в которых он участвовал, должны быть ликвидированы.

Такого содержательного эффекта можно достичь многими способами. Здесь естественно учесть то, что пользователи лишены возможности непосредственно изменять «сеть» (например, явными присваиваниями этому массиву), они могут к нему добираться только посредством объявленных в спецификации пакета процедур и функций. Наша задача как реализаторов пакета – обеспечить согласованный внешний эффект объявленных услуг (при этом внутренний эффект процедур и функций можно варьировать).

Другими словами, действие процедуры «удалить» на массив «сеть» должно быть таким, чтобы функции узел_есть и все_связи выдали результаты, согласованные с содержательным представлением об отсутствии узла в сети. Один вариант реализации – присвоить false соответствующему полю «включен» и подправить поле «связан» во всех узлах, с которыми был связан удаляемый узел. Другой вариант – в этой процедуре поле «связан» не подправлять, но изменить реализацию функции все_связи так, чтобы перед выдачей результата она приводила поле «связан» в соответствие с полем «включен».

Это и есть варианты упоминавшихся выше дисциплин работы с сетью.

Рациональность выбора одного из вариантов неочевидна. Если часто удаляют узлы и редко просят все их связи, может оказаться выгодным второй вариант,

иначе – первый. Оценить частоту запросов – дело непростое. Поэтому возможность менять реализацию (подстраиваясь к условиям эксплуатации), не меняя внешнего эффекта, может оказаться очень важной.

Обратите внимание, не спроектировав представления данных, мы не могли начать проектировать процедуры. А теперь видим, что проектирование данных может зависеть от дисциплины взаимодействия операций. В этом – одно из проявлений **принципа единства основных абстракций**, о котором мы еще поговорим.

Выберем первый вариант реализации.

```
procedure удалить(X : in узел) is
begin
   сеть(X).включен := false;
   for i in 1..сеть(X).связан.число loop
      чистить(X,сеть(X).связан.узлы (i));
   end loop;
end;
```

Понадобилась процедура «чистить», которая должна убрать в узле, указанном вторым параметром, связь с узлом, указанным первым параметром.

```
procedure чистить(связь : узел, в_узле : узел) is
begin
   for i in 1..сеть(в_узле).связан.число loop
      if сеть(в_узле).связан.узлы (i) = связь then
         переписать(в_узле, после => i);
      end if;
   end loop;
end чистить;
```

Осталось спроектировать процедуру «переписать» – она должна переписать связи в указанном узле, начиная с номера «после», и уменьшить на единицу общее число связей этого узла.

```
procedure переписать(в_узле : in узел, после : in индекс_узла) is
   запись:связи renames сеть(в_узле).связан;
begin
   запись.число := запись.число - 1;
   for j in после..запись.число loop
      запись.узлы(j) := запись.узлы(j+1);
   end loop;
end переписать;
```

Здесь мы впервые воспользовались ОБЪЯВЛЕНИЕМ ПЕРЕИМЕНОВАНИЯ (renames), чтобы сократить имена и сделать их более наглядными. Этот же прием можно было применять и раньше. Напомним, что о диагностике ошибок мы пока не заботимся (предполагается, что перед применением процедуры «удалить» всегда применяется функция узел_есть, чтобы не было попытки удалить несуществующий узел).

«Запись» – это имя объекта типа «связи» (объекта сеть(в_узле).связан), локальное для процедуры «переписать». Общий вид ОБЪЯВЛЕНИЯ ПРОЦЕДУРЫ:

```
<спецификация процедуры>  is
<локальные объявления>;
begin
   <операторы>;
end процедуры;
```

Оборот for j in <диапазон> – это ОБЪЯВЛЕНИЕ УПРАВЛЯЮЩЕЙ ПЕРЕМЕННОЙ ЦИКЛА, область действия которой – от объявления до конца цикла. Внутри БАЗИСНОГО ЦИКЛА (от loop до end loop) j считается постоянной. Если диапазон пуст (это бывает, когда его правая граница меньше левой), базисный цикл не выполняется ни разу. Иначе он выполняется при всех последовательных значениях j из указанного диапазона, если только выполнение всего оператора цикла не будет досрочно завершено оператором выхода (exit).

В нашем случае все имена узлов из массива «узлы» с индексами от «после+1» до «число» перемещаются на позиции с предыдущим индексом. В результате массив «узлы» содержит все старые связи, кроме вычеркнутой, а их общее количество предварительно скорректировано (уменьшено на 1).

Шаг 3.7. Содержательный эффект процедуры «связать» также очевиден: она применима к включенным в сеть узлам; после ее применения узлы считаются связанными.

Снова можно было бы реализовать такой эффект по-разному. Выберем следующий способ, учитывающий конкретные реализации остальных наших процедур: в запись о каждом из аргументов процедуры «связать» будем добавлять указание о связи с другим аргументом.

По-прежнему не будем заботиться о диагностике ошибок, когда связей оказывается слишком много (больше макс_связей). Но если два узла просят связать вторично, то будем такой запрос игнорировать. Следует учесть также, что требование связать узел с самим собой вполне законно.

```
procedure связать(X, Y: in узел) is
begin
   if not есть_связь(X, Y) then
      установить_связь(X, Y);
      if X /= Y then
         установить_связь(У, X);
      end if;
   end if;
end связать;
```

Мы ввели вспомогательную функцию есть_связь с очевидным эффектом (возможно, ее полезно и пользователю предоставить) и вспомогательную процедуру установить_связь, которая призвана вносить изменения в массив «узлы» своего первого аргумента. (**Ключевое слово not – это знак отрицания (унарная логическая операция)**).

Продолжим детализацию.

```
function есть_связь(X, Y : узел) return BOOLEAN is
запись : связи renames сеть(X).связан;
```

```
begin
   for i in 1..запись.число loop
      if запись.узлы(i) = Y then
         return true;
      end if;
   end loop;
   return false;
end есть_связь;

procedure установить_связь(откуда, куда : in узел) is
   запись : связи renames сеть(откуда).связан;
begin
      запись.число := запись.число+1;
      запись.узлы(запись.число) := куда;
end установить_связь;
```

Таким образом, количество связей увеличивается на единицу, и в качестве последней связи записывается имя узла «куда».

> **Вопрос.** Нельзя ли переименование указать вне процедур и функций, чтобы не повторять его?
>
> **Подсказка.** В переименовании участвуют динамические параметры.

Итак, все услуги реализованы. Осталось выписать полное тело пакета. Для экономии места и времени позволим себе не выписывать объявления процедур и функций полностью, обозначая пропуски многоточием.

Обратите внимание на порядок следования объявлений. Он существен, соответствует правилу последовательного определения. Но оно касается лишь вспомогательных объявлений, введенных в теле пакета. Имена из спецификации пакета считаются объявленными ранее.

```
package body управление_сетью is
   type запись_об_узле is
   record
      включен : BOOLEAN : = false;
      связан: связи;
   end record;
сеть : array (узел) of запись_об_узле;
function узел_есть(X : узел) return BOOLEAN is
         ......
         function все_связи(X : узел) return связи is
         ......
         procedure вставить (X : in узел) is
         ......
         procedure переписать(в_узле : in узел, после : in индекс_узла)
         ......
         procedure чистить(связь : узел, в_узле:узел)  is
         ......
         procedure удалить(X : in узел) is
         ......
```

```
        function есть_связь(X, Y : узел) return BOOLEAN is
        ......
        procedure установить_связь(откуда, куда : in узел) is
        ........
        procedure связать(X, Y : in узел) is
        ........
end управление_сетью;
```

Подчеркнем, что тип запись_об_узле, объект «сеть», процедуры «переписать», «чистить», «установить_связь», функция «есть_связь» недоступны пользователю, так как объявлены в теле, а не в спецификации пакета.

Третий шаг детализации завершен. Осталась прокомментировать полученный результат.

2.7. Принцип раздельного определения, реализации и использования услуг (принцип РОРИУС)

Итак, мы написали три сегмента: спецификацию пакета управление_сетью, процедуру построение_сети и тело пакета управление_сетью. Важно понимать роли этих сегментов в жизненном цикле программы.

В них воплощен **принцип раздельного определения, реализации и использования услуг (РОРИУС)**. По существу, это рациональное применение абстракции на различных этапах проектирования.

Проектируя определение пакета, отвлекаемся от деталей его возможного использования и вариантов реализации.

Проектируя использование пакета, отвлекаемся от деталей определения и тем более реализации.

Проектируя реализацию, отвлекаемся от несущественного (с точки зрения реализации) в определении и использовании.

> **Упражнение.** Приведите конкретные примеры деталей, несущественных при определении, реализации и использовании соответственно.

Каждая названная абстракция (определение, использование, реализация) представлена своим материальным воплощением – отдельным модулем. Ведь каждый из трех сегментов является модулем – законченным продуктом интеллектуальной деятельности в том смысле, что его можно записать в библиотеку и (при наличии документации) использовать без помощи автора и без жесткой связи с остальными модулями.

Три наших модуля, однако, не являются полностью независимыми. Центральным служит, конечно, модуль определения, то есть спецификация пакета. Оставляя спецификацию неизменной, можно выбирать варианты реализации (тело па-

кета), не заставляя изменять использование (процедуры, аналогичные процедуре «построение_сети»). И это только благодаря тому, что реализация защищена от несанкционированного доступа при использовании – из процедуры построение_сети нельзя непосредственно добраться, например, до массива «сеть» и нарушить дисциплину его эксплуатации операциями пакета. С другой стороны, никакое изменение реализации (согласованное со спецификацией и содержательным внешним эффектом объявленных услуг) не в состоянии повлиять на какие-либо характеристики использования, кроме ресурсоемкости (расхода времени, памяти и других ресурсов). Наконец, можно строить произвольные, одновременно существующие и развивающиеся, по-разному использующие модули, не тратя ресурсов на однажды уже определенные и реализованные услуги.

Таким образом, рассмотренные разновидности модулей, воплощающие абстракции определения, использования и реализации услуг, удовлетворяют важнейшую технологическую потребность – проектировать, использовать и хранить программы по частям, отдельными модулями.

2.8. Принцип защиты абстракций

Кроме РОРИУС, в связи с только что отмеченной потребностью необходимо указать на еще один важный принцип – **принцип защиты абстракций** (от разрушения). Одно из его проявлений в Аде – доступ к телу пакета исключительно через имена, объявленные в спецификации. Именно благодаря этому пользователь получает абстрактный объект – сеть, которой может пользоваться, но не может ее разрушить. Абстрактность сети проявляется в том, что пользователь не знает деталей ее представления и реализации доступа. Принцип защиты абстракций обслуживают и другие конструкты Ады (в частности, приватные типы).

Обратите внимание, что, создав пакет управление_сетью, мы в сущности спроектировали ПОЯ, построив подходящую модель предметной области (модель сети связи). Тем самым показали, как пользоваться Адой в качестве базового ЯП. При этом средством абстрактного определения ПОЯ служит спецификация пакета, средством конкретизации – тело пакета, а средством защиты – невидимость из использующих сегментов имен, объявленных в теле пакета.

Глава 3

Важнейшие абстракции: данные, операции, связывание

3.1. Принцип единства и относительности трех абстракций 70
3.2. Связывание 71
3.3. От связывания к пакету 72
3.4. Связывание и специализация ... 74
3.5. Принцип цельности 79

3.1. Принцип единства и относительности трех абстракций

Планируя поведение исполнителя (составляя программу), мы в конечном итоге планируем его действия над некоторыми объектами. Необходимо обозначить, во-первых, конкретный объект – данное, во-вторых, конкретное действие – операцию и, в-третьих, обозначить конкретные условия, при которых нужно указанное действие совершить над указанным объектом (то есть обозначить условия связывания данных и операций). Таким образом, в описании акта исполнителя выделяются составляющие, играющие три различные роли: данных, операций и связывания. Подчеркнем, что в каждом конкретном акте (когда указана операция, указаны данные и созрели условия для связывания) три названные роли неразрывны.

Однако программировать было бы невозможно, если бы не удалось разорвать единство этих ролей и рассматривать данные, в нужной степени абстрагируясь от конкретных операций; рассматривать операции, в нужной степени абстрагируясь от конкретных данных, над которыми они выполняются; и, наконец, рассматривать связывание, в нужной степени абстрагируясь от данных и операций, которых оно касается.

Полученное путем такой абстракции понятие операции отражает активное начало в поведении исполнителя, понятие данного – пассивное начало, а понятие связывания – управляющее (организующее) начало – отражает всевозможные виды управления.

Первые две роли выделяют и обсуждают чаще. Однако значение связывания никак не меньше. Как будет показано, оно отражает не только разнообразные способы управления, но и разнообразные способы конкретизации в программировании.

Важно понимать, что указанные три роли полностью различны лишь в рамках конкретного акта исполнителя. Когда же речь идет о каком-либо отдельно взятом объекте, то в зависимости от ситуации или точки зрения он вполне может выступать в различных ролях (иногда – в любой из этих ролей). В этом смысле указанные роли(как и любые другие) относительны. Например, знак «+» чаще всего воспринимается как обозначение операции сложения. Вместе с тем он выступает в роли данного, когда фигурирует как элемент формулы, которую переводят в польскую инверсную запись. Но этот же знак связывает два операнда формулы, между которыми он помещен, предвосхищая их совместное участие в одном акте исполнителя. Этот акт будет выполнен при условии, что управление достигло именно рассматриваемого знака «+».

Относительность и единство (взаимосвязь, взаимозависимость) трех выделенных ролей – глубокая закономерность. Проявляется она, в частности, в том, что для достаточно развитого оформления каждой абстракции привлекаются и две другие. Однако в этом случае они играют подчиненную роль, роль обслуживающего средства,роль инструмента для выделения и (или) реализации основной абстракции.

Рассмотрим ряд примеров, подтверждающих принцип относительности и единства выделенных абстракций. Покажем, что он не только объясняет появление и смысл языковых конструктов, но и указывает перспективы их развития.

Так, хороший стиль программирования предполагает, что когда вводят операционную абстракцию (процедуру, функцию, операцию), то явно указывают характеристики данных, к которым ее можно применять (с которыми ее можно связывать). Например, специфицируют параметры процедуры (как мы и поступали в задаче об управлении сетью). Такую спецификацию можно назвать спецификацией операции по данным извне (со стороны использования). Когда реализуют введенную операционную абстракцию (проще говоря, программируют процедуру или функцию), то также указывают характеристики данных (на этот раз используемых, подчиненных, желательно локальных; в нашем примере это был тип запись_об_узле). Такую спецификацию можно назвать спецификацией операции по данным изнутри (со стороны реализации).

Вопрос. Когда полезна такая спецификация?

Подсказка. Ведь это, в сущности, заказ ресурсов, который необходим для их рационального распределения, особенно в режиме разделения ресурсов между параллельными процессами.

В современных ЯП (в том числе в Аде), когда вводят абстракцию данных (переменнную, тип переменнных), то указывают класс операций, связываемых с этой абстракцией (определяют абстрактный тип данных, АТД). Это можно назвать спецификацией данных по операциям извне (со стороны использования). В нашем примере это пять операций – вставить, удалить, связать, все_связи и узел_есть, характеризующих доступ к абстрактной переменной «сеть». Из соображений симметрии следовало бы рассматривать и данные (абстракции данных), для реализации которых нужны внутренние (локальные) операции. Это была бы спецификация данных по операциям изнутри.

Вопрос. Зачем могут понадобиться такие спецификации?

Подсказка. Представьте «живые» данные, отражающие состояние взаимодействующих процессов. Какие операции потребуются, чтобы их реализовать? Какой должна быть среда, в которую можно перенести такие данные? Конкретный пример – монитор Хоара-Хансена, о котором пойдет речь в главе об асинхронных процессах. Для его реализации требуются операции над сигналами, семафорами или рандеву.

3.2. Связывание

Так как выделение связывания в качестве самостоятельной абстракции менее традиционно, полезно рассмотреть его подробнее и проиллюстрировать плодотворность этого выделения нетривиальными применениями.

В самом общем понимании связывание неотличимо от установления соответствия, сопоставления, отношения. Однако в применении к программированию

нас интересует определенный набор выразительных средств, управляющих связыванием и пригодных для реализации на компьютерах. Поэтому, говоря о связывании здесь, мы подразумеваем аппарат, действующий в рамках существующей или потенциальной системы программирования и предназначенный для организации взаимодействия операций и данных.

Постараемся показать, во-первых, что абстракция связывания помогает с единых позиций понимать кажущиеся совершенно различными сущности, и, во-вторых, что выделение такой абстракции помогает увидеть возможные направления развития ЯП.

Для начала подчеркнем, что в качестве составляющих единого аппарата связывания в системе программирования естественно рассматривать редактор связей, загрузчик, оптимизатор, компилятор, интерпретатор и другие системные средства, предназначенные для подготовки программы к выполнению. Ведь все эти средства участвуют в различных этапах окончательного связывания конкретных операций с конкретными операндами.

Итак, можно говорить о связывании загрузочных модулей для последующего совместного использования. Такая работа выполняется редактором связей. Можно говорить о связывании аргументов подпрограммы с ее телом для последующего совместного выполнения. Такое связывание обеспечивается вызовом и заголовком подпрограммы. Вполне разумно говорить и о связывании отдельных компонент объектной программы в процессе ее трансляции с ЯП. Выполняется оно, естественно, транслятором.

Отметим важный аспект. Связывание может распадаться на этапы, выполняемые на различных стадиях подготовки того конкретного акта исполнителя, ради которого это связывание в конечном итоге осуществляется. Например, транслировать можно за несколько проходов; связать загрузочные модули можно частично редактором связей, частично загрузчиком; при обращении к программе часть работы по связыванию можно выполнить при обращении к подпрограмме, часть – при выполнении ее тела (именно распределением работы по связыванию отличаются различные способы вызова параметров – наименованием, значением, ссылкой и др.).

По-видимому, все эти примеры хорошо известны. Но общая концепция связывания способна привести и к совершенно новому понятию, отсутствующему в традиционных ЯП.

3.3. От связывания к пакету

Из общего курса программирования известно, что такое контекст. Известно также, что модуль – это программа, рассчитанная на многократное использование в различных контекстах (и для этого соответствующим образом оформленная). В традиционных ЯП контекст задается обычно совокупностью объявлений (описаний) некоторого блока или подпрограммы и связывается с телом блока текстуально, физическим соединением тела и контекста. Но если модуль рассчитан на различные контексты, то и контекст, естественно, может оказаться пригодным

для работы с различными модулями. Следовательно, хорошо бы и контекст оформлять по таким правилам, чтобы его не нужно было выписывать каждый раз, а можно было использовать как модуль, связывая с телом блока, например, во время трансляции, Подобной категории модулей ни в Алголе, ни в Фортране, ни в Паскале нет. Впервые такой модуль появился в языке Симула-67 и был назван «классом». В Аде его аналог назван «пакетом».

Рассмотрим подробнее путь к пакету на конкретном примере.

В общем курсе программирования при изучении структур данных знакомят с совокупностью понятий, позволяющих работать со строками. Например, определяют представление строк одномерными массивами и предоставляют несколько операций над строками (например, в-строку, из-строки и подстрока). Спрашивается, каким образом оформить это интеллектуальное богатство так, чтобы им было удобно пользоваться? Алгол-60 или Паскаль позволяет записать соответствующие объявления массивов и процедур, а тем самым сделать их известными многим программистам. Совокупность указанных объявлений массивов, переменных и процедур, выписанная в начале блока, позволяет в теле блока работать в сущности на языке, расширенном по сравнению с Алголом-60 (понятием строчных переменных и набором операций над такими переменными).

Но вот мы знаем (изучили) эти объявления и хотим ими воспользоваться (например, запрограммировать и запустить универсальный нормальный алгоритм Маркова, как нам предлагают авторы того же курса). Алгол-60 заставляет нас переписать в свою программу все нужные объявления. Но это и труд, и ошибки, и время, и место на носителях. На практике, конечно, во многих реализациях Алгола-60 есть возможность обращаться к библиотеке, где можно хранить объявления функций и процедур (но не переменных и массивов). Однако целостного языкового средства, обслуживающего потребность делать доступным расширение языка, однажды спроектированное и полностью подготовленное к использованию, нет. Другими словами, не выделена абстракция связывания компонент потенциально полезного контекста. Нет ее ни в Паскале, ни в Фортране, хотя общие объекты последнего – намек на движение в нужном направлении.

Как уже сказано, впервые нужная абстракция была осознана и оформлена соответствующим конструктом в языке Симула-67. Основная идея в том, что совокупность объявлений можно синтаксически оформить (в качестве «класса»), предварив их ключевым словом class и снабдив индивидуальным именем. Так можно получить, например, класс с именем обработка_строк, в котором будут объявлены одномерный массив и процедуры для работы с этим массивом как со строкой символов. Чтобы воспользоваться такими объявлениями (в совокупности!), достаточно перед началом программы указать в качестве приставки имя нужного класса. Объявления из такого класса считаются выписанными в фиктивном блоке, объемлющем создаваемую программу (то есть доступны в ней). Например, программу нормального алгоритма достаточно предварить приставкой обработка_строк.

В первом приближении основная идея ПАКЕТА совпадает с идеей класса – это также совокупность объявлений, снабженная именем и пригодная для исполь-

зования в качестве «приставки». Однако в понятии «пакет» воплощены и другие важнейшие идеи, о которых уже шла речь. Подчеркнем, что к новым понятиям нас привела общая концепция связывания.

> **Вопрос.** Чем идея пакета (модуля-контекста) отличается от идеи простого копирования контекста? От идеи макроопределений?
>
> **Подсказка.** Важно, когда происходит связывание, а также чего и с чем. Кроме того, не забывайте об управлении доступом к контексту.

3.4. Связывание и специализация

Не только отдельные языковые конструкты обязаны своим возникновением тому, что связывание было осознано как самостоятельная абстракция. На его основе возникло целое направление в программировании – так называемое конкретизирующее программирование (как было отмечено, связывание обобщает основные виды конкретизации). Когда говорят о конкретизирующем программировании, часто приводят такой пример.

Рассмотрим операцию «**» возведения основания x в степень n. Если понятно самостоятельное значение связывания, то легко представить себе ситуацию, когда с операцией «**» уже связан один операнд и еще не связан другой. С точки зрения итогового возведения в степень, такая ситуация запрещена – еще нельзя совершить запланированный акт поведения (операнды не готовы). Но если понимать связывание как многоэтапный процесс подготовки этого акта, то рассматриваемая ситуация может соответствовать одному из этапов этого процесса. Более того, на аналогичном этапе связывания могут задерживаться целые классы таких процессов. Это повторяющееся следует выделить, обозначить и применить (пользуемся одним из важнейших общих принципов абстрагирования – принципом обозначения повторяющегося). Так получается целый ряд одноместных операций («1**»,«2**»,«3**»...) при фиксированном основании и ряд одноместных операций («**1»,«**2»,«**3»...) при фиксированном показателе степени. Но, например, операцию «**3» можно реализовать просто как x*x*x, что короче, проще и эффективней общей программы для «**».

Таким образом и возникает идея универсального конкретизатора, который по параметрической программе (например, «**») и некоторым уже связанным с ней аргументам строит (потенциально более эффективную) конкретизированную программу (например, «**3»). Если такой конкретизатор удастся построить для некоторого класса программ, то возникнет надежда обеспечить целую проблемную область эффективными и надежными, «по происхождению» правильными программами. Ведь исходная параметрическая программа предполагается сделанной исключительно тщательно (во всяком случае, правильно) – при таком ее широком назначении на нее не жалко усилий.

В настоящее время конкретизирующее программирование интенсивно развивается и у нас, и за рубежом. Конкретизатор в литературе называют иногда специализатором, а также смешанным вычислителем (за то, что он проводит вычисления и над данными, и над программами).

Отметим, что любые языковые конструкты можно при желании считать частью аппарата связывания. Ведь с их помощью аргументы программы связываются с ее результатами на той или иной стадии обработки ее текста. Воспользуемся этим наблюдением, чтобы продемонстрировать еще одно применение аппарата связывания, – уточним терминологию и укажем некоторые перспективы в теории трансляции. Это же наблюдение положено в основу трансформационного подхода к программированию [3].

3.4.1. Связывание и теория трансляции

Основной результат настоящего раздела: такие программы, как компилятор и суперкомпилятор (генератор компиляторов), могут быть формально получены из интерпретатора ЯП с помощью подходящего связывания.

Ключевая идея: следует применить особый вид связывания, обобщающий обычный вызов функции таким образом, что часть параметров функции оказывается связанной со своими аргументами, а остальные остаются пока несвязанными и служат параметрами остаточной функции. **Остаточной** называют функцию, вызов которой с оставшимися аргументами эквивалентен вызову исходной функции с полным набором аргументов. Такой вид связывания называют специализацией.

Число аргументов функции несущественно, важно лишь отделить связываемые раньше и позже. Поэтому для уяснения основной идеи достаточно рассмотреть функции с двумя аргументами.

Введем понятие «универсального специализатора». Вслед за Бэкусом назовем **формой** функцию высшего порядка, то есть функцию, аргумент и (или) результат которой также представляет собой некоторую функцию. «Универсальный специализатор» s – это форма, которая по произвольной функции двух переменных F(X,Y) и заданному ее аргументу x0 выдает в качестве результата функцию одного аргумента s(F,x0), такую, что для всех допустимых значений параметра Y справедливо определяющее соотношение

(**) s(F,x0) (Y) = F(x0,Y),

так что s(F,x0) – это и есть остаточная функция после связывания первого параметра функции F с аргументом x0.

Покажем, как получить объявленный основной результат. Допустим, что все рассматриваемые функции и формы реализованы подходящими программами. Сохраним для этих программ те же обозначения. Так что s(F,x0) можно теперь считать программой, полученной по исходной программе F с помощью программы s.

Замечание. Важно понимать, что о качестве получаемых специализированных (остаточных) программ в определении универсального специализатора ничего не сказано. Тривиальное преобразование программ может состоять, например, в том, что в остаточной программе просто содержится вызов вида F(x0,Y).

Упражнение. Запишите тривиальную остаточную программу на одном из известных вам ЯП.

Рассмотрим теперь язык программирования L и его интерпретатор i. С одной стороны, i – это такая программа, что для всякой правильной программы p на языке L и исходных данных d:

i(p,d) = r,

где r – результат применения программы p к данным d. Другими словами, программа i реализует семантику языка L – ставит в соответствие программе p результат ее выполнения с данными d. С другой стороны, i – это форма от двух аргументов (а именно так называемая ограниченная аппликация – она применяет свой первый аргумент-функцию p ко второму аргументу d, причем пригодна только для программ из L).

Интерпретатор может быть реализован аппаратно, то есть быть отдельным устройством, предназначенным для выполнения программ на L. Однако для нас интереснее случай, когда интерпретатор реализован программой. Программа эта написана, конечно, на каком-то языке программирования M. Будем считать, что M отличен от L. Программная реализация интерпретатора интересна именно потому, что в этом случае интерпретатор представлен написанным на языке M текстом-программой, и вполне можно ожидать, что в общем случае из этого текста можно систематическими преобразованиями получать другие программы. Например, программы компилятора и суперкомпилятора, также написанные на языке M.

Мы намерены делать это посредством специализатора s. Для определенности будем считать, что программа-специализатор s также написана на языке M, применима к текстам программ, написанным на M, и выдает в качестве результатов программы, написанные все на том же языке M.

Специализация интерпретатора. Посмотрим, что собой представляет s(i,p), то есть во что специализатор s превращает интерпретатор i после его связывания с конкретной программой p. (Ведь i – форма от двух аргументов, так что специализатор s к ней применим; при этом в соответствии со смыслом s с i связывается первый аргумент интерпретатора – p, а второй остается свободным параметром.) Применяя (**), получаем

s(i,p)(d) = i(p,d) = r.

Обратите внимание, чтобы выписать результат специализатора, нужно «передвинуть» функциональные скобки на позицию вправо и опустить символ специализатора.

Другими словами, s(i,p) – это такая программа p’, которая после применения к данным d дает результат r. Следовательно, p’ эквивалентна программе p. Но p’ написана уже на языке M, а не на L! Следовательно, p’ – это перевод программы p на язык M. Итак, связав интерпретатор (написанный на языке M) с исходной программой на языке L, получили ее перевод на M.

Кратко это можно выразить так: специализация интерпретатора по программе дает ее перевод.

> Подумайте, что в общем случае можно сказать о качестве полученного перевода – скорости работы, объеме памяти; а что – о скорости перевода?

Специализация специализатора. Итак, при различных i специализатор дает переводы с разных языков. Нетрудно теперь догадаться, что при фиксированном i специализатор s представляет собой компилятор с языка L на язык M. Ведь, как мы видели, в этом случае он по заданной p получает ее перевод p'. Действительно, посмотрим, что такое s(s,i)? Вновь применяя (**), получаем

s(s,i)(p) = s(i,p).

Но ведь s(i,p) – это p', перевод программы p на язык M! Так что s(s,i) (написанный на M) – это компилятор KLM с языка L на язык M.

Кратко выразим это так: **специализация специализатора по интерпретатору дает компилятор**. Или еще короче: автоспециализация по интерпретатору дает компилятор.

Снова есть повод подумать о возможном качестве компилятора и затратах на его получение в общем случае.

Двойная автоспециализация. Специализатор может выступать и в роли суперкомпилятора. Ведь по заданному интерпретатору i (который можно считать описанием языка L) специализатор выдает компилятор с языка L на язык M. Действительно, посмотрим, что такое s(s,s)? Опять применяя (**), получаем

s(s,s)(i) = s(s,i).

Но ведь s(s,i) = KLM! Так что s(s,s) – это действительно суперкомпилятор над языком M (в свою очередь написанный на M).

Кратко выразим это так: **двойная автоспециализация дает суперкомпилятор**.

Вопрос. Нельзя ли получить что-либо интересное тройной автоспециализацией?

Подсказка. А что, если подставлять различные воплощения специализатора s?

Три последовательных применения специализатора удобно наглядно выразить следующей серией соотношений:

s(s,s)(i)(p)(d) = s(s,i)(p)(d) = s(i,p)(d) = i(p,d) = r.

Другими словами, s(s,s) воспринимает описание языка L (то есть i) и выдает компилятор s(s,i), который, в свою очередь, воспринимает исходную программу p на языке L и выдает ее перевод s(i,p), который уже воспринимает исходные данные d и выдает результат r.

Таким образом, мы убедились, что абстракция связывания (точнее, частичное связывание) позволяет с единых позиций рассмотреть важнейшие понятия теории трансляции и вывести полезные закономерности. Именно связав i с p, получили перевод; связав s с i, получили компилятор; связав s с s – суперкомпилятор.

Строго говоря, мы имеем здесь дело не с суперкомпилятором, а с более универсальной программой.

Вопрос. В чем это проявляется?

Приведенные выше соотношения называют соотношениями Футамуры-Турчина. Способ их изложения позаимствован у С. А. Романенко.

Замечание (о сущности трансляционных понятий). Хотя непосредственное практическое значение соотношений Футамуры-Турчина пока проблематично, они помогают увидеть заманчивые перспективы, а также четче выделять понятия.
Действительно, обычно отличие, например, интерпретации от компиляции формулируют несколько расплывчато. Говорят, что интерпретатор воспринимает исходную программу вместе с исходными данными и выполняет ее последовательно, «шаг за шагом», в соответствии с операционной семантикой языка L. Операционной называют семантику, выраженную через последовательность действий исполнителя, соответствующую каждому тексту на L.

Вопрос. Можно ли иными средствами задать семантику ЯП? Предложите свои средства.

Написание интерпретаторов на машинных или ранее реализованных языках – хорошо известный, естественный и для многих целей удобный способ реализации ЯП. Для некоторых из них (Лисп, Апл, Бейсик) – единственный способ полной реализации. Это справедливо для всех языков, в которых программа может меняться в процессе исполнения, – только «шаг за шагом» и можно уследить за таким изменением.
Когда говорят о компиляции, подразумевают перевод всей программы как целого, без учета конкретных исходных данных, с исходного языка L на объектный язык M. С конкретными исходными данными исполняется уже результат подобного перевода.
Такого рода содержательные различия, конечно, существенны, однако значительная их часть улавливается на формальном уровне, нам теперь вполне доступном. Ведь интерпретатор – это форма с двумя аргументами, а компилятор – с одним. Интерпретатор – это (ограниченный) аппликатор, а компилятор – это преобразователь программ (сохраняющий их смысл).
Обратите внимание: приведен пример пользы от рассмотрения языковых концепций (связывания) с математической позиции.
С другой стороны, важно понимать, что формальные преобразования специализатора в компилятор и суперкомпилятор не отражают некоторых содержательных аспектов этих понятий. Обычно компилятор применяют ради повышения скорости работы переведенных программ по сравнению с интерпретацией. Специализатор же в общем случае может выдать остаточную программу, состоящую, в сущности, из интерпретатора и обращения к нему. В таком случае неоткуда ждать выигрыша в скорости. При попытках «оптимизировать» такую программу за счет раскрытия циклов и т. п. она может стать непомерно длинной. Аналогичные соображения касаются и суперкомпилятора. Тем не менее в указанном направлении получены обнадеживающие результаты для частных видов специализаторов [4, 5].
Не до конца улавливается приведенными соотношениями и сущность компиляции. Она – в переводе на другой язык, на котором может оказаться вовсе невозможно или очень невыгодно писать интерпретатор исходного языка L (например, это невозможно делать на небольшой встроенной бортовой машине). А ведь в наших соотношениях все программы (кроме p) написаны на объектном языке M. Сказанное не означает, что в подобных случаях непригодна математическая позиция. Просто нужны и другие математические модели компиляции. Например, проекционная, где компилятор рассматривается как реализация проекции (отображения языка L на M), а не как специализация написанного на M интерпретатора (последнего может и не существовать).

На этом закончим обсуждение связывания как самостоятельной абстракции. Как видим, оно оказалось весьма емким, глубоким понятием, взаимодействующим со многими концепциями программирования.

3.5. Принцип цельности

Считая достаточно обоснованной самостоятельную ценность каждой из трех выделенных абстракций, продемонстрируем на их примере один весьма общий принцип проектирования, который назовем **принципом цельности**. Его называют также принципом концептуальной целостности.

Суть принципа цельности – в том, что детали проекта в идеале должны быть следствием относительно небольшого числа базисных, ключевых решений. Другими словами, в цельном проекте большинство деталей можно предсказать, зная базисные решения. Содержательно это означает, что проект выполнен на основе цельной концепции, единого замысла, а не представляет собой нагромождения случайностей.

Покажем, как принцип цельности проявляется на трех уровнях рассмотрения программного проекта, два из которых – языковые.

Первый уровень – собственно программа. Сначала несколько совсем общих соображений. Проектирование – это сочетание абстракции и конкретизации (принимая конкретное проектировочное решение, тем самым одновременно вводят абстракции нижнего уровня, предназначенные для реализации принятого решения, – вспомните появление новых имен при проектировании пакета управление_сетью). Цельная концепция в идеале должна воплощаться согласованными абстракциями, а отсутствие таковой проявляется в их несогласованности.

Согласованность (абстракций) понимается как удобство их совместного использования для удовлетворения определяющих потребностей.

Возвратимся к трем выделенным абстракциям. Заметим, что иметь с ними дело приходится независимо от того, насколько сознательно они выделяются в технологическом цикле проектирования программ. Покажем, как принцип цельности позволяет выработать естественные критерии качества языковых конструктов.

> Следующие ниже соображения носят весьма общий, почти философский характер. Это естественно, так как рассматривается один из общих принципов «философии программирования». Вместе с тем получаются вполне осязаемые критерии и оценки.

В соответствии с **принципом технологичности** (который справедлив не только для ЯП, но и для создаваемых с их помощью программ) выделяемые абстракции призваны обслуживать определенные технологические потребности. Проявим критерии цельности ЯП с точки зрения потребностей пошаговой детализации.

Применяя эту технологию, следует исходить из хорошо понятной постановки задачи. Однако в понятной постановке задачи компонент мало. Следовательно, они содержательные, емкие, имеющие непосредственную связь с сутью решаемой

задачи. Но тогда и операнды у этих операций обладают теми же свойствами. И их связывание должно опираться на средства доступа к таким емким операциям и данным. По мере детализации операций менее емкими становятся и операнды, и средства связывания.

Итак, в процессе пошаговой детализации свойства данных, операций и связывания должны на каждом шаге быть взаимно согласованными по степени детализации.

> **Упражнение.** Проверьте это утверждение на нашем примере с сетями. Обратите внимание на возможность вводить содержательные понятия, не слишком беспокоясь пока о возможности их реализации средствами ЯП.

Второй уровень – средства программирования. Следовательно, чтобы обеспечить технологические потребности пошаговой детализации, языковые конструкты должны обслуживать согласование указанных свойств на каждом шаге детализации. Мы пришли к важному критерию качества языковых конструктов: в хорошем ЯП конструкты, обслуживающие определение и использование каждой из упомянутых абстракций, должны быть согласованы по степени управления детализацией.

> **Упражнение.** Приведите примеры нарушения этого требования в известных вам ЯП.

> **Подсказка.** В Паскале функции вырабатывают только скалярный результат – нет прямого средства сопоставить «емкой» функции согласованную по «емкости» структуру данных. Например, нельзя определить функцию все_связи. Уже упоминалось отсутствие развитых средств связывания с контекстом.

Третий уровень – средства развития. Подчеркнем важный момент. Содержательные абстракции на каждом шаге детализации зависят от решаемой задачи. Как уже говорилось, для работы в конкретных прикладных областях создаются ПОЯ. Это и есть задача, решаемая с помощью базового языка. Решать ее естественно также методом пошаговой детализации (иногда его называют методом абстрактных машин). Например, мы создали абстрактную машину, работающую с сетью. Ее команды – наши пять операций доступа к сети. При реализации этой машины используется (но не оформлена нами явно) машина удаления связей. Никлаусу Вирту принадлежит глубокая мысль о том, что истинное назначение ЯП – предоставить средства для ясного и точного определения абстрактных машин. Следовательно, в базовых языках должны быть согласованы между собой и средства развития всех трех разновидностей абстракций. Другими словами, в ЯП, претендующем на универсальность, принцип цельности в идеале призван работать при проектировании как базиса ЯП, так и средств его развития.

> **Упражнение.** Приведите примеры нарушения этого требования в известных вам ЯП.

> **Подсказка.** Проще всего это сделать по отношению к связыванию – в традиционных ЯП этот аппарат развит слабо. Легко ли, например, на Паскале определить ПОЯ, в котором возможно раздельно определять заголовки и тела процедур, а связывать их по мере необходимости?

Итак, несмотря на свой весьма абстрактный характер (скорее, благодаря ему), принцип цельности обнаруживает «точки роста» ЯП, намечает тенденции их развития: в частности, от языков (готовых) программ к языкам собственно программирования, позволяющим с исчерпывающей полнотой управлять связыванием компонент программы [3].

3.5.1. Принцип цельности и нормальные алгоритмы

Принцип цельности носит неформальный, почти эстетический характер. Способность оценивать уровень цельности языков и программ приходит только с опытом. Чтобы лучше прочувствовать этот принцип, попробуем оценить на его основе язык нормальных алгоритмов – модель Маркова. В этой модели всякая программа представляет собой линейную последовательность однородных операций (подстановок) над линейной последовательностью однородных данных (символов). Связывание также однородно – просматривается последовательность операций и последовательность данных, конкретная операция-подстановка связывается с подходящей последовательностью данных. Затем эта операция выполняется, и происходит новый цикл связывания.

Отметим очевидную согласованность всех трех абстракций. Но эта согласованность обслуживает не пошаговую детализацию, а простоту исследования свойств нормальных алгоритмов (именно к этому и стремился их изобретатель). Вместе с тем очевидно, что как основные абстракции, так и средства развития в этой модели не удовлетворяют потребностям пошаговой детализации.

Вопрос. В чем это проявляется?

Таким образом, качество ЯП не определяется критерием цельности самим по себе. Влияние этого критерия на оценку качества ЯП зависит от того, какие технологические потребности признаются определяющими.

Известный тезис нормализации утверждает, что всякий алгоритм можно заменить эквивалентным нормальным алгоритмом. Но важно хорошо понимать смысл слова «можно» в этом тезисе. Можно заменить, если принять абстракцию потенциальной осуществимости, отвлечься от таких «несущественных деталей», как необходимые для этого ресурсы. Любой, кто писал нормальные алгоритмы, прекрасно понимает, что ни одной реальной программы непосредственно в исходной модели Маркова нельзя даже написать – она практически наверняка будет неправильной, и отладить ее будет невозможно в обозримое время (даже если предположить сколь угодно высокую скорость выполнения самих марковских подстановок). Ведь все выделяемые при программировании абстракции, как данных, так операций и связывания, нужно подразумевать или хранить вне программы. Поэтому единственный разумный путь к практической осуществимости программирования на языке нормальных алгоритмов – моделировать на этом языке другой, более совершенный в технологическом отношении язык.

3.5.2. Принцип цельности и Ада. Критерий цельности

Как видно на примере Ады, в более современных ЯП принцип согласования абстракций (как между собой, так и с важнейшими технологическими потребностями) осознан и учтен в гораздо большей степени. Взглянем на шаги с 3.1 по 3.7 на стр. 60–66 с точки зрения потребности согласовывать абстракции.

Выделив на шаге 3.1 операционную абстракцию – функцию все_связи, мы немедленно ощутили потребность обозначить классы возможных аргументов и результатов этой функции, не занимаясь их детальной проработкой. Если бы приходилось работать на Фортране, Алголе-60 или Бейсике, сделать это оказалось бы невозможным – непосредственно подходящих предопределенных типов данных в этих ЯП нет, а возможность строить новые типы также отсутствует. Скорее всего, пришлось бы нарушить естественный порядок детализации и сначала придумать способ представлять «связи» некоторым массивом, а затем учесть, что в этих ЯП функции не вырабатывают результаты-массивы (только скаляры), и представить нужную абстракцию не функцией, а процедурой. Важно понимать, что такого рода отклонения запутывают логику программы, провоцируют ошибки, затрудняют отладку и т. п.

Лучшее, что можно сделать в этом случае, – выйти за рамки используемого ЯП и фиксировать шаги детализации на частично формализованном псевдокоде. О применении такого псевдокода в структурном подходе к программированию на классических ЯП можно прочитать, например, в [6].

В Аде мы смогли провести шаг детализации полностью в рамках языка. Причем, вводя операционную абстракцию, были вынуждены воспользоваться средствами определения абстракций другого рода – абстракций данных. Точнее говоря, на шаге 3.1 мы воспользовались лишь тем, что в Аде можно вводить новые абстракции данных и можно вводить для этих абстракций подходящие названия.

Обратите внимание, мы ввели не названия отдельных объектов данных (только так и можно в классических ЯП), а именно названия целых классов (типов) обрабатываемых объектов.

Определения типов мы также имели возможность вводить по шагам, вполне аналогично тому, как в классических ЯП вводят операционные абстракции, выделяя нужные процедуры. На шаге 3.1 обозначили новый тип «связи»; на шаге 3.5 уточнили его строение, но потребовались названия типов число_связей и перечень_связей. На шагах 3.6 и 3.7 уточнили строение этих типов, но остались неопределенными макс_узлов и макс_связей. Наконец, уточнили их характеристики и даже значения.

Четкость пошаговой детализации поддерживалась языковыми средствами связывания. Можно было не заботиться о реализации процедур и функций – ее можно определить позже, в теле пакета – средства связывания обеспечат согласованное использование спецификаций и тел процедур. В классических ЯП так поступить нельзя, пришлось бы опять выходить за рамки языка или нарушать поря-

док детализации и выписывать тела процедур (а это не всегда можно сделать, еще не зная структуры используемых данных).

Итак, должно быть видно, как принцип согласования основных абстракций (между собой и с потребностями пошаговой детализации) воплощен в Аде. Во-первых, согласованность основных абстракций действительно требовалась, и, во-вторых, Ада необходимые выразительные средства предоставляет.

Принцип цельности дает основания ввести критерий технической оценки языка, который можно назвать **критерием цельности**: **язык тем лучше, чем ближе он к идеалу с точки зрения принципа согласования абстракций**. Ясно, что с точки зрения технологии пошаговой детализации Ада превосходит классические ЯП по этому критерию.

Упражнение. Пользуясь критерием цельности, оцените другие известные ЯП.

Глава 4

Данные и типы

4.1. Классификация данных 86
4.2. Типы данных 88
4.3. Регламентированный доступ
и типы данных 98
4.4. Характеристики, связанные
с типом. Класс значений,
базовый набор операций 106
4.5. Воплощение концепции
уникальности типа. Определение
и использование типа в Аде
(начало) 107
4.6. Конкретные категории типов 108
4.7. Типы как объекты высшего
порядка. Атрибутные функции 129
4.8. Родовые (настраиваемые)
сегменты 131
4.9. Числовые типы (модель
числовых расчетов) 133
4.10. Управление операциями 137
4.11. Управление
представлением 138
4.12. Классификация данных
и система типов Ады 141
4.13. Предварительный итог
по модели А 143

4.1. Классификация данных

Рассматривая три выделенные роли в акте исполнителя, мы подчеркивали, что с ролью данных ассоциируется пассивное начало. Это не означает и не требует полной пассивности объекта, выступающего в роли данного, а лишь его относительной пассивности с точки зрения рассматриваемого акта поведения того исполнителя, планирование поведения которого нас интересует. Тот же объект, с другой точки зрения, может быть активным и даже сам выступать в роли исполнителя. В качестве примера можно привести задачу управления асинхронными процессами, когда осуществляющий управление исполнитель вправе рассматривать эти (активные) процессы как данные, на которые направлено его управляющее воздействие.

Данными обычно считают любые обрабатываемые объекты независимо от их внутренней природы. Одна и та же категория объектов в одном ЯП может выступать в роли данных, а в другом может быть запрещена. Так, процедуры могут быть данными в Паскале и Алголе-68 (их можно передавать в качестве значений, присваивать компонентам других объектов), но не в Аде.

Данные различаются по многим признакам.

Во-первых, данные можно классифицировать по содержательным ролям, которые они играют в решаемой задаче. Очень заманчиво было бы уметь явно отражать в программе результаты такой классификации, с тем чтобы сделать ее доступной как исполнителю, так и читателю программы. По существу, это прогнозирование поведения определенных объектов данных (например, прогноз о том, что переменные А и В никогда не могут быть операндами одного и того же сложения). Прогноз такого рода облегчает понимание программы и создает предпосылки для автоматического содержательного контроля.

Возможность отражать содержательную классификацию данных отсутствует в большинстве классических ЯП. В Аде сделаны шаги в нужном направлении. Например, мы различали типы «узел», индекс_узла и число_связей, хотя все они в конечном итоге представлены целыми числами.

Во-вторых, данные различаются по своему внутреннему строению, структуре, характеру связей своих составляющих. Например, массивы, таблицы, списки, очереди. С этой точки зрения важен способ доступа к составляющим данных. Классификация данных по способу доступа к составляющим обычно имеется в виду, когда говорят о структурах данных. Классификация данных по их структуре есть в том или ином варианте почти во всех ЯП. В современных ЯП чаще всего выделяются массивы и записи. И то, и другое можно считать частным случаем таблиц, которые служат ключевой структурой, например в МАСОНе [10] и его последующих модификациях.

В-третьих, данные различаются по своей изменчивости. Например, в некоторых случаях известен диапазон возможных изменений или известно, что данное вообще не должно меняться. Прогнозирование поведения такого рода встречается только в относительно новых ЯП, начиная с Паскаля.

В-четвертых, данные могут различаться по способу своего определения. Их свойства могут быть предопределены (то есть определены автором ЯП) или же определены программистом с помощью языковых средств. В последнем случае в идеале это такие средства, которые позволяют программисту по существу определить новый язык, обогащая исходный.

> Как правило, предопределенные объекты и свойства не могут быть изменены программистом и в этом смысле надежно защищены от искажений. У программиста должна быть возможность принять меры к тому, чтобы вновь введенные им абстракции были неотличимы от предопределенных. Это еще одна формулировка принципа защиты абстракций.
> Если защита обеспечена, то на каждом шаге обогащения ЯП появляется полная возможность действовать так, как будто в распоряжении программиста появился виртуальный исполнитель для сконструированного уровня абстракции. Подчеркнем, что при этом детализация осуществляется от задачи к реализации (сверху вниз), а создание виртуальных машин – в общем случае от реальной машины к задаче (снизу вверх). Аппарат для определения данных, ориентированный на принцип защиты абстракций, имеется только в новейших ЯП, в частности в Аде, Модуле-2, последних версиях Паскаля и др.

В-пятых, данные могут различаться по своему представлению на более низком уровне абстракции (на реализующей виртуальной машине, в терминах реализующей структуры данных, по классу необходимых для реализации ресурсов, по объему и дисциплине использования памяти и т. п.). Например, для чисел может требоваться одно, два или несколько слов в зависимости от нужной точности вычислений, память для данных одной категории может выделяться в некотором стеке (например, для локальных данных блоков) или в так называемой куче (для элементов динамически изменяемых списковых структур), для некоторых данных разумно выделять самую быструю память (например, быстрые регистры для переменной цикла). Это еще одна форма прогнозирования поведения объектов – запрос для них подходящих ресурсов. Классификация с точки зрения представления встречается практически во всех ЯП, ориентированных на эффективное использование ресурсов машины, в частности в языке Си.

В-шестых, данные могут различаться по применимым операциям (внешним свойствам), определяющим возможности данного играть определенные роли или вступать в определенные отношения с другими объектами программы.

Например, если данное представляет собой число, символ, указатель, задачу, очередь, стек, то в каждом из этих случаев к нему применим определенный набор операций, у него имеется определенный набор атрибутов, характерных для данных именно этого класса, и т. п. Чтобы не было путаницы с первым фактором классификации (по содержательным ролям), подчеркнем, что переменная для хранения числа апельсинов может отличаться от переменной для хранения числа яблок по содержательной роли, но не отличаться по применимым операциям. А вот объекты типов «узел» и индекс_узла различаются по применимым операциям (**по каким?**).

Наконец, **в-седьмых**, данные могут различаться по характеру доступа к ним. Одни данные считаются общедоступными, другие могут использоваться только определенными модулями или при определенных условиях и т. п.

В заключение подчеркнем, что указанные факторы ортогональны. Классификация не претендует на полноту, но позволит ориентироваться, в частности, в системе управления данными в Аде. Дополнительные факторы классификации предложены, например, в языке Том [7].

4.2. Типы данных

Классификация данных присутствует в каждом ЯП. В Алголе-60 она отражена в системе классов и типов (классы – процедуры, метки, простые переменные, массивы, переключатели; типы – целый, вещественный, логический), в Фортране – также в системе классов и типов (классы имен – массив, переменная, внутренняя функция, встроенная функция, внешняя функция, подпрограмма, переменная и общий блок; типы – целый, вещественный, двойной точности, комплексный, логический, текстовый). В более современных ЯП имеется тенденция полнее отражать классификацию данных в системе типов, а само понятие типа данных меняется от языка к языку.

Система типов в ЯП – это всегда система классификации денотатов (в общем случае и данных, и операций, и связываний; возможно, и других сущностей). Задачи такой классификации зависят от назначения языка, а также от других причин (отражающих, в частности, специфику ЯП как явления не только научно-технического, но и социального).

На систему типов в Аде влияет, конечно, специфика встроенных систем программного обеспечения (в особенности требование повышенной надежности, эффективности объектной программы и относительное богатство ресурсов инструментальной машины). Но некоторые свойства этой системы явно предписаны техническими требованиями заказчика и не могли быть изменены авторами языка.

Таким образом, излагаемые ниже принципы построения системы типов в Аде нужно воспринимать как интересный и в целом дееспособный вариант классификации обрабатываемых данных, но отнюдь не как окончательное единственно верное решение. Элегантная теория типов предложена А. В. Замулиным и воплощена в ЯП Атлант [8, 9].

4.2.1. Динамические, статические и относительно статические ЯП

Некоторые свойства объекта и связи с другими объектами остаются неизменными при любом исполнении его области действия (участка программы, где этот объект считается существующим). Такие свойства и связи называются статическими. Их можно определить по тексту программы, без ее исполнения.

Например, в Паскале тип объекта (целый, вещественный, логический) – одно из статических свойств. Сама область действия объекта – по определению статическое его свойство. Связь двух объектов по свойству принадлежать одной области действия – статическая связь. Свойство объекта при любом исполнении облас-

ти действия принимать значения только из фиксированной совокупности значений – статическое свойство. Исчерпывающий перечень применимых к объекту операций – статическое свойство.

Другие свойства и связи изменяются в процессе исполнения области действия. Их называют динамическими.

Например, конкретное значение переменной – динамическое свойство. Связь формального параметра с конкретным фактическим в результате вызова процедуры – динамическая связь. Размер конкретного массива с переменными границами – динамическое свойство.

Часто статические и динамические характеристики называют соответственно характеристиками периода компиляции (периода трансляции) и периода выполнения, подчеркивая то обстоятельство, что в период компиляции исходные данные программы недоступны и, следовательно, динамические характеристики известны быть не могут. Известны лишь характеристики, извлекаемые непосредственно из текста программы и тем самым относящиеся к любому ее исполнению (то есть статические характеристики).

Однако деление на статические и динамические характеристики иногда оказывается слишком грубым. Например, размер массива в Алголе-60 может в общем случае изменяться при различных исполнениях его области действия, однако при каждом конкретном исполнении этот размер зафиксирован при обработке объявления (описания) массива и в процессе исполнения области действия изменяться не может. Так что это и не статическая характеристика, и вместе с тем не столь свободно изменяемая, как, например, значение компоненты массива в Алголе-60, которое можно изменить любым оператором присваивания. Такие характеристики, которые могут меняться от исполнения к исполнению, но остаются постоянными в течение одного исполнения области действия объекта, будем называть относительно статическими.

Иногда пользуются и еще более тонкой классификацией характеристик по фактору изменчивости. Например, связывают изменчивость не с областью действия объекта, а с периодом постоянства других его избранных характеристик (выделяемого объекту пространства, связи с другими объектами и т. п.).

Уровень изменчивости характеристик допустимых денотатов – одно из важнейших свойств ЯП. Одна крайняя позиция представлена концепцией неограниченного (образно говоря, «разнузданного») динамизма, когда по существу любая характеристика обрабатываемого объекта может быть изменена при выполнении программы. Такая концепция не исключает прогнозирования и контроля, но не связывает их жестко со структурой текста программы.

Неограниченный динамизм присущ не только практически всем машинным языкам, но и многим ЯП достаточно высокого уровня. Эта концепция в разной степени воплощена в таких динамических ЯП, как Бейсик, Алл, Лисп, отечественных ИНФ и Эль-76 [10, 11]. Идеология и следствия динамизма заслуживают отдельного изучения.

Другая крайняя позиция выражена в стремлении затруднить программисту всякое изменение характеристик денотатов. Вводя знак, нужно объявить характе-

ристики денотата, а использование знака должно соответствовать объявленным характеристикам. Конечно, «неограниченной» статики в программировании добиться невозможно (**почему?**). Так что всегда разрешается менять, например, значения объявленных переменных.

Зато остальные характеристики в таких статических ЯП изменить трудно. Обычно стремятся к статике ради надежности программ (за счет дополнительной избыточности, при обязательном объявлении характеристик возникает возможность дополнительного контроля) и скорости объектных программ (больше связываний можно выполнить при трансляции и не тратить на это времени в период исполнения).

Вместе с тем сама по себе идея объявления характеристик (прогнозирования поведения) и контроля за их инвариантностью требует создания, истолкования и реализации соответствующего языкового аппарата. Поэтому статические ЯП, как правило, сложнее динамических, их описания объемнее, реализации – тяжеловеснее. К тому же надежды на положительный эффект от статики далеко не всегда оправдываются. Тем не менее среди массовых языков индустриального программирования преобладают статические. Раньше это частично можно было объяснить трудностями эффективной реализации динамических ЯП. Сейчас на первое место выходит фактор надежности, и с этой точки зрения «старые» статические ЯП оказываются «недостаточно статическими» – аппарат прогнозирования и контроля у них связан скорее с требуемым распределением памяти, чем с другими характеристиками поведения, существенными для обеспечения надежности (содержательными ролями, изменчивостью значений, применимыми операциями и т. п.). С этой точки зрения интересен анализ возможностей динамических ЯП, в частности Эль-76, содержащийся в [11].

Ада принадлежит скорее к статическим, чем к динамическим ЯП, ее можно назвать языком относительно статическим с развитым аппаратом прогнозирования-контроля. Концепция типа в Аде предназначена в основном для прогнозирования-контроля статических характеристик. Ее дополняет концепция подтипа, предназначенная для прогнозирования-контроля относительно статических характеристик. В дальнейшем мы будем рассматривать эти концепции вместе, считая концепцию подтипа составной частью концепции типа.

4.2.2. Система типов как знаковая система

Постановка задачи. На стр. 86–87 мы выделили семь факторов, характеризующих данные: роль в программе, строение, изменчивость, способ определения, представление, доступ, применимые операции. ЯП как знаковая система, предназначенная для планирования поведения исполнителя (в частности, для планирования его манипуляций с данными), может как иметь, так и не иметь специальные понятия и конструкты, позволяющие программисту характеризовать данные.

Крайняя позиция – полное или почти полное отсутствие таких средств. Пример – нормальные алгоритмы Маркова и другие модели, которые мы еще рассмотрим, а также Лисп, Форт, Апл, где нельзя охарактеризовать данные ни по од-

ному из семи факторов. Конечно, эти факторы существенны независимо от применяемого ЯП. Просто при проектировании программы на ЯП без специальных средств классификации данных программист вольно или невольно использует для характеристики данных внеязыковые средства (держит «в уме», отражает в проектной документации и т. п.).

Кое-что из такой «внешней» классификации находит воплощение и в программе. Но если находит, то часто в такой форме, что программиста не могут проконтролировать не только компьютер, но и коллеги-программисты (да и он сам делает это, как правило, с трудом).

Указанная крайняя позиция игнорирует потребность прогнозирования-контроля. Она характерна для ранних машинных ЯП и в чистом виде в современном программировании не встречается.

Вспоминая пример с пошаговой детализацией и принцип согласования абстракций, можно почувствовать, что тенденция к развитым средствам описания данных – естественная тенденция в современных ЯП, ориентированных на создание надежных и эффективных программ с ясной структурой.

Допустим, что эта тенденция осознана. Возникает следующая задача – какие ключевые концепции должны быть положены в основу средств описания данных? Если угодно, как построить «язык в языке», знаковую систему для характеристики данных в ЯП?

Первый вариант: перечни атрибутов. По-видимому, первое, что приходит в голову, – ввести свои средства для характеристики каждого фактора и сопровождать каждый объект перечнем характеристик.

Примерно так сделано в ПЛ/1. Объявляя переменную, в этом ЯП можно перечислить ее характеристики (так называемые «атрибуты») по многим факторам (основание системы счисления, способ представления, способ выделения памяти, структура и способ доступа к компонентам, потребуется ли печатать значения и т. п.).

Такая знаковая система, как показывает опыт, вполне дееспособна, однако с точки зрения ясности и надежности программ оставляет желать лучшего.

Во-первых, довольно утомительно задавать длинные перечни атрибутов при объявлении данных. Из-за этого в ПЛ/1 придуманы даже специальные «правила умолчания» атрибутов (это одна из самых неудачных и опасных с точки зрения надежности концепция ПЛ/1, к тому же этих правил много, они сложны, так что запомнить их невозможно).

Во-вторых, один перечень атрибутов – у данного, а другой (тоже достаточно длинный) перечень – у формального параметра процедуры. Может ли такое данное быть фактическим параметром? Чтобы это понять, нужно сравнить оба перечня, потратив время и рискуя ошибиться (компьютер ошибиться не рискует, но ему это тоже дорого обходится). И дело не только (и не столько) в самом переборе атрибутов, сколько в сложности правил, определяющих применимость процедуры к данному.

Итак, запомнив, что у проблемы два аспекта – прогноз (описание) и контроль (проверка допустимости поведения данных), поищем другие решения.

Отметим, что по отношению к ЯП мы сейчас колеблемся между технологической и авторской позициями, задевая семиотическую.

Второй вариант: структурная совместимость типов. Воспользуемся принципом обозначения повторяющегося. Заметили, что часто приходится иметь дело с перечнями атрибутов? Значит, нужно обозначить повторяющийся перечень некоторым именем и упоминать это имя вместо самого перечня. Следовательно, нужен языковый конструкт для объявления таких имен.

Естественно считать, что имя обозначает не только сам перечень атрибутов, но и класс данных, обладающих объявленными характеристиками. Мы пришли к понятию типа данных как класса объектов-данных, обладающих известными атрибутами. А искомый языковой конструкт – это объявление типа данных. Его назначение – связывать объявляемое имя с указанным перечнем атрибутов данных. Подчеркнем, что основой классификации данных остается перечень характеристик-атрибутов, а имя типа лишь обозначает соответствующий перечень.

Теперь, чтобы характеризовать данное, не нужно умалчивать атрибуты, как в ПЛ/1, можно коротко и ясно обозначать их одним именем. Кажется, совсем просто, но это шаг от ПЛ/1 к Алголу-68, в котором очень близкая к изложенной концепция типа данных.

С проблемой прогнозирования мы справились, а как с проблемой контроля? По-прежнему нужно сравнивать перечни атрибутов. Другими словами, здесь мы никак не продвинулись. Как справиться с проблемой? Ведь для контроля поведения данных (например, контроля совместимости аргументов с параметрами) недостаточно знать имена типов: если имена совпадают, все ясно, а вот когда не совпадают, нужно проверять совместимость характеристик типов.

Проблема структурной совместимости типов (иногда говорят, проблема структурной эквивалентности типов, потому что простейшее правило совместимости – эквивалентность атрибутов) в общем случае очень сложна, может оказаться даже алгоритмически неразрешимой.

Дело в том, что как только возникают имена типов, естественно их применять и в перечнях атрибутов. Например, при определении комбинированного типа указываются типы полей, при определении процедурного типа – типы параметров процедуры и т. п. Но в таком случае имя типа может оказаться связанным уже не с единственным перечнем атрибутов, а с классом таких перечней (возможно, бесконечным, например, описывающим свойства рекурсивных структур-списков). Другими словами, допустимые для каждого типа перечни атрибутов определяются, например, контекстно-свободной грамматикой (БНФ). Так что проблема эквивалентности типов сводится к проблеме эквивалентности контекстно-свободных грамматик, которая в общем случае алгоритмически неразрешима.

Третий вариант: именная совместимость типов. Поистине блестящее решение состоит в том, чтобы полностью избавиться от проблемы структурной совместимости, «чуть-чуть» подправив концепцию типа, сделав центральным понятием не атрибуты, а имя типа. При этом типы с разными именами считаются разными и в общем случае несовместимыми (если программист не ввел соответствующих операций преобразования типов). Другими словами, забота о содержательной со-

вместимости характеристик объектов полностью перекладывается на программиста, определяющего типы (что вполне естественно). Если теперь потребовать, чтобы каждый объект данных был связан ровно с одним типом, то и прогнозировать, и проверять – одно удовольствие! Нужно сказать, что объект будет вести себя так-то, обозначаем стиль (характер) его поведения именем типа из имеющегося набора типов. Нет подходящего – объявляем новый. Нужно проверить совместимость – сравниваем имена типов. Просто, ясно и быстро.

Но наше «чуть-чуть» – в историческом плане шаг от Алгола-68 с его структурной совместимостью типов к Аде с ее именной совместимостью через Паскаль, где именная совместимость принята для всех типов, кроме диапазонов (для них действует структурная совместимость).

4.2.3. Строгая типизация и уникальность типа

Априорная несовместимость типов, названных разными именами, вместе с идеей «каждому объекту данных – ровно один тип» образуют концепцию типа, которую мы назовем концепцией уникальности типа (или просто уникальностью типа). Это ключевая концепция аппарата типов в Аде. Сформулируем правила (аксиомы) **уникальности типа**:

1. Каждому объекту данных сопоставлен один и только один тип.
2. Каждому типу сопоставлено одно и только одно имя (явное или неявное). Типы с неявными именами называются анонимными и все считаются различными.
3. При объявлении каждой операции должны быть явно указаны (специфицированы) имена типов формальных параметров (и результата, если он есть).
4. Различные типы априорно считаются несовместимыми по присваиванию и любым другим операциям.

Очень близкая концепция в литературе часто называется **строгой типизацией** [26], а ЯП с такой концепцией типа – строго типизированными. От уникальности типа строгая типизация отличается возможным ослаблением четвертой аксиомы. Поэтому мы ввели специальный термин «уникальность типа» для «совсем строгой» типизации.

4.2.4. Критичные проблемы, связанные с типами

Остался маленький вопрос. Прогнозировать – легко, проверять – легко. А легко ли программировать? Как увязать несовместимость типов, обозначенных разными именами, с естественным желанием иметь операции, применимые к объектам разных типов (полиморфные операции). Ведь если связать с формальным параметром операции некоторый тип, то к объектам других типов эта операция окажется неприменимой.

Назовем отмеченную проблему проблемой полиморфизма операций. Напомним, что полиморфизм операций широко распространен в математике, жизни, программировании. Операция «+» применима к целым числам, вещественным числам, матрицам, комплексным числам, метрам, секундам и т. п.

Проблема полиморфизма возникла, как только мы взглянули на уникальность типа со стороны операций. Но неприятности поджидают нас и со стороны данных. Вполне естественны ситуации, когда в процессе работы программы один и тот же объект выступает в разных ролях, обладает различными характеристиками. Например, число яблок может превратиться в число людей, взявших по яблоку; буфер, работавший в режиме очереди, может начать функционировать в режиме стека; у массива может измениться размерность и т. п.

Каким должен быть единственный (уникальный) тип такого объекта? Если он отражает сразу много ролей, то не придем ли мы к отсутствию контроля (ведь пока буфер работает как очередь, к нему нужно запретить обращаться как к стеку, и наоборот)? Если же только одну роль, то как ему попасть в другую (ведь разные типы априорно несовместимы)? Назовем выделенную проблему янус-проблемой (объект ведет себя как двуликий Янус, даже «многоликий»).

4.2.5. Критичные потребности и критичные языковые проблемы

Несколько драматизируем ситуацию и временно представим, что найти решение проблемы полиморфизма не удалось. И вот мы в роли программиста, который старается следовать хорошему стилю создания программ.

Допустим, что к его услугам тип «целый», тип «комплексный», тип «матрица». Ему нужно предоставить пользователю возможность складывать объекты любого из названных типов. Хороший стиль программирования требует ввести операционную абстракцию, позволяющую пользователю игнорировать несущественные детали (в данном случае – особенности каждого из трех типов данных) и действовать независимо от них. Попросту говоря, нужно ввести единую (полиморфную) операцию сложения. Если такой возможности ЯП не предоставит, то у программиста может оказаться единственный разумный вариант – избегать пользоваться таким ЯП. Аналогичным ситуациям нет числа. Например, есть возможность рассчитать потребности взвода, роты, батальона, полка. Требуется ввести абстракцию – «рассчитать потребности подразделения» и т. п.

Мы пришли к понятию критичной технологической потребности. Технологическая потребность называется **критичной** для ЯП в некоторой ПО, если язык не может выжить в этой ПО без средств удовлетворения этой потребности. Проблему удовлетворения критичной потребности назовем **критичной языковой проблемой**.

4.2.6. Проблема полиморфизма

Допустим, что критичность проблемы полиморфизма для Ады осознана. Как же ее решать?

По-видимому, самым естественным было бы ввести «объединяющий» тип (например, «операнд_сложения» или «боевое_подразделение»), частными случаями которого были бы исходные типы. И определить нужную операцию для объединяющего типа. Но что значит «частными случаями»? Ведь в соответствии с концепцией уникальности каждый объект принадлежит только одному типу. Если это «взвод», то не «боевое_подразделение»! Так что в чистом виде эта идея не проходит.

> Нечто подобное можно реализовать с помощью так называемых ВАРИАНТНЫХ типов, но каждый вариант должен быть выделен соответствующим значением дискриминанта, причем заботиться об этом должен пользователь такой «квазиполиморфной» операции. Поэтому подобное решение можно рассматривать лишь как суррогат.

Вот если бы к услугам программиста уже был тип «боевое_подразделение», а ему понадобилось ввести новые операции именно для взводов, то (при условии, что у объектов «боевое_подразделение» есть поле «вид») можно было бы объявить, например:

```
type T – взвод  is new боевое_подразделение (вид => взвод);
```

Теперь «взвод» – это уже знакомый нам ПРОИЗВОДНЫЙ тип с РОДИТЕЛЬСКИМ типом боевое_подразделение. К его объектам применимы все операции, применимые к боевому_подразделению. Но можно теперь объявить новые операции, применимые только к «взводам», то есть к «боевым_подразделениям», в поле «вид» которых – значение «взвод». Итак, это решение проблемы полиморфизма «сверху вниз», то есть нужно заранее предусмотреть частные случаи нужного типа.

Основной вариант решения проблемы полиморфизма, предлагаемый Адой, – это так называемое ПЕРЕКРЫТИЕ операций. Идея состоит в том, что связь между вызовом операции и ее объявлением устанавливается не по одному только имени операции, а по так называемому «профилю» (имени операции с учетом типов операндов, типа результата и даже имен формальных параметров (если вызов по ключу)). Другими словами, идея перекрытия – в том, что денотат знака определяется не по самому знаку, а с привлечением его ограниченного контекста.

Например, в одной и той же области действия можно объявить две функции:

```
function потребности (подразделение:взвод) return расчет;
function потребности (подразделение:рота) return расчет;
```

Для каждой функции нужно написать свое тело. Если теперь объявить объекты

```
A:взвод;
B:рота;
```

то вызов потребности(A) будет означать выполнение тела первой функции, а вызов потребности(B) – второй. Для пользователя же «видна» единственная операционная абстракция «потребности», применимая к объектам и типа «взвод», и типа «рота», то есть полиморфная функция.

> **Упражнение.** Укажите дополнительный контекст знака функции в приведенном примере.

Конечно, такое решение требует своего заголовка и своего тела для каждого варианта допустимых типов параметров, но это и есть плата за полиморфизм. К тому же все не так страшно, как может показаться. Ведь самое главное, что пользователь получает в точности то, что нужно, – полиморфную операцию, сохраняя полный контроль над использованием объектов в соответствии с их типами во всех остальных случаях. Это полиморфизм «снизу вверх», когда частные случаи операции можно добавлять (причем делать это несложно).

Вопрос. Как это сделать?

4.2.7. Янус-проблема

Вспомним, как возникла янус-проблема. Мы пришли к концепции уникальности, желая упростить контроль. И потеряли возможность иметь объекты, играющие одновременно разные роли.

Но система типов в каждой программе – это некоторая классификация. Одна из простейших и самая распространенная классификация – иерархическая. Хорошо известный пример такой классификации – классификация животных и растений по типам, классам, отрядам, семействам, родам и видам. Подчеркнем, что при этом каждое животное (классифицируемый объект) играет сразу несколько ролей (он и представитель вида, и представитель рода, и представитель семейства, и т. п.).

К характеристикам типа (общим для всех животных этого типа) добавляются специфические характеристики класса (общие только для выбранного класса, а не для всего типа), затем добавляются характеристики отряда и т. д., вплоть до характеристик вида.

Еще сложней ситуация, когда классификация не иерархическая. Человек – одновременно сотрудник лаборатории, отдела, института и т. п.; жилец в квартире, доме, микрорайоне и т. п.; подписчик газеты, муж, брат, сват, любитель бега и т. п. Если нужно написать на Аде пакет моделирование_человека, то как уложиться в концепцию уникальности типа?

Напомним, что проблема возникла именно потому, что мы хотим прогнозировать и контролировать различные роли объектов. Если игнорировать проблему прогнозирования-контроля, то исчезнет и янус-проблема. Как в Алголе-60 – везде массивы целых, и представляй их себе в любой роли (ведь эти роли – вне программы!).

Полного и изящного решения янус-проблемы Ада не предлагает – этого пока нет ни в одном ЯП. Ближе всего к идеалу – объектно-ориентированные ЯП.

Но можно выделить три основных свойства Ады, направленных на решение янус-проблемы. Каждое из них по-своему корректирует концепцию уникальности, а вместе они образуют практически приемлемое решение.

Эти средства – ПРОИЗВОДНЫЕ ТИПЫ + ПРЕОБРАЗОВАНИЯ типов + понятие ОБЪЕКТА ДАННЫХ.

Производные типы. Мы уже видели, что объявление производного типа указывает родительский тип и определяет, что объекты производного типа могут

принимать лишь подмножество значений, допустимых для объектов родительского типа. Вместе с тем для объектов производного типа можно определить новые операции, неприменимые в общем случае к объектам родительского типа. Но ведь это связь старшей и младшей категорий в иерархической классификации.

Если, например, определен тип «млекопитающие», то его производным может стать тип «хищные», его производным – тип «кошки», его производными – типы «сибирские_кошки» и «сиамские_кошки». При этом все уменьшается совокупность допустимых значений (в «хищные» не попадают «коровы», в «кошки» – «собаки», в «сибирские_кошки» – «львы») и добавляются операции и свойства (млекопитающие – операция «кормить молоком»; хищные – «съедать животное»; кошки – «влезать на дерево»; сибирские кошки – «иметь пушистый хвост»), причем всегда сохраняются операции и свойства всех родительских типов, начиная с так называемого БАЗОВОГО ТИПА, не имеющего родительского типа.

Итак, производные типы решают проблему полиморфизма «сверху вниз» – это одновременно частное решение янус-проблемы.

Вопросы. Почему это решение названо решением «сверху вниз»? Сравните с предложенным ранее решением «снизу вверх». Почему это лишь частное решение янус-проблемы?

Подсказка. Для полиморфизма: неполиморфная операция «надстраивается» над независимыми типами, а типы заранее строятся «сверху вниз» так, что операция над объектами «старшего» типа оказывается применимой и к объектам «младшего». Для янус-проблемы: существенно, что при объявлении типа «млекопитающие» нужно знать об атрибутах потенциальных производных типов, иначе негде спрятать «пушистый хвост» – подробнее об этом в разделе о наследовании с критикой Ады; вспомните также о пакете модель_человека – там не спасет иерархия типов.

Преобразования типов. Считается, что каждое объявление производного типа неявно вводит и операции ПРЕОБРАЗОВАНИЯ ОБЪЕКТОВ из родительского типа в производный и обратно. При этом перейти от родительского типа к производному можно только при выполнении объявленных ограничений на значения объекта, а обратно – всегда. Можно написать процедуру, например «кормить_молоком» для типа «млекопитающие», и применять ее и к «телятам», и к «котятам», и к «львятам». При связывании аргумента с параметром выполняется подразумеваемое преобразование типа, и процедура применяется к аргументу как к «млекопитающему». Но нельзя применять процедуру «влезть_на_дерево» к «корове» – только к «кошке».

Возможны и явно определяемые программистом преобразования типа. В них нет ничего удивительного – это просто функции, аргументы которых одного типа, а результаты – другого. Их естественно считать преобразованием типа, если они сохраняют значение объекта в некотором содержательном смысле. Так, можно написать преобразование из типа «очередь» в тип «стек», сохраняющее содержимое объекта. К результату такого преобразования можно применять операции «втолкнуть», «вытолкнуть», определенные для стеков, а к аргументу нельзя. Подчеркнем, что написать преобразование можно далеко не в каждом контексте –

нужно иметь возможность «достать» содержимое аргумента и «создать» содержимое результата.

Объекты данных. Осталось уточнить понятие «объект данных». Напомним: уникальность требует, чтобы типы в программе образовывали разбиение объектов данных, то есть чтобы каждый объект данных попадал в точности в один тип. Другими словами, типы не пересекаются и объединение объектов всех типов – это и есть множество всех объектов данных программы.

Но это значит, что к объектам данных следует относить только сущности, с которыми связан тип (в описании ЯП или в программе). Таковыми в Аде служат, во-первых, изображения предопределенных значений (изображения чисел, символов и логических значений); во-вторых, переменные, постоянные, динамические параметры, выражения. В Аде не относятся к объектам данных процедуры, функции, типы, сегменты, подтипы. Имена у них есть, а типов нет. Их нельзя присваивать и передавать в качестве аргументов процедур и функций – это их основное отличие от объектов данных.

4.2.8. Критерий содержательной полноты ЯП. Неформальные теоремы

В заключение этого раздела обратим внимание на способ решения критичных технологических проблем. Каждый раз требовались и чисто языковые средства, и методика применения этих средств.

Для проблемы полиморфизма это языковое средство (перекрытие операций) плюс методика определения серии операции с одним названием. Для янус-проблемы это снова языковое средство (производные типы) плюс методика определения системы производных типов для реализации нужной классификации данных. Наличие и языкового средства, и методики служит доказательством неформальной теоремы существования решения критичной проблемы.

С авторской позиции исключительно важен критерий качества, который можно назвать критерием полноты ЯП: автор ЯП должен уметь доказывать существование решения всех известных критичных проблем. Другими словами, уметь доказывать неформальную теорему содержательной полноты ЯП по отношению к выбранной ПО. Вполне возможно, что позже будут найдены другие, более удачные решения критичных проблем, однако это уже не столь принципиально с точки зрения жизнеспособности ЯП.

4.3. Регламентированный доступ и типы данных

Начав с общей потребности прогнозировать и контролировать, мы пришли к общей идее приемлемого решения – к идее уникальности типа, обозначенного определенным именем. Теперь нужно разобраться со средствами, позволяющими свя-

зать с именем типа технологически значимые характеристики данных (характер доступа, строение, изменчивость, представление и т. п.).

В соответствии с принципом технологичности эти средства обслуживают определенные потребности жизненного цикла комплексного программного продукта. Чтобы лучше почувствовать, что это за потребности, продолжим нашу серию примеров. Ближайшая цель – подробнее рассмотреть средства управления режимом доступа к данным.

4.3.1. Задача моделирования многих сетей

Постановка задачи. Допустим, что пользователям понравился пакет управление_сетью и они заказывают более развитые услуги. Одной сети мало. Нужно сделать так, чтобы пользователи могли создать столько сетей, сколько им потребуется, и по отношению к каждой могли воспользоваться любой из уже привычных операций (вставить, связать и т. п.). Вместе с тем больше всего пользователи оценили именно надежность наших программных услуг, гарантию целостности сети. Это важнейшее свойство необходимо сохранить.

> Пользователи предъявляют внешнее требование надежности услуг, а возможно (если они достаточно подготовлены), и целостности создаваемых сетей. Наша задача как реализаторов комплекса услуг – перевести внешние требования (потребности) на язык реализационных возможностей.
> Эти потребности и возможности относительны на каждом этапе проектирования. Скажем, потребность в надежности реализуется возможностью поддерживать целостность. Потребность в целостности реализуется возможностью обеспечить (строго) регламентированный доступ к создаваемым сетям.

Итак, будем считать, что технологическая потребность пользователя – в том, чтобы иметь в распоряжении класс данных «сети», иметь возможность создавать сети в нужном количестве и получать регламентированный доступ к каждой созданной сети.

Варианты и противоречия. Казалось бы, такую потребность несложно удовлетворить – достаточно предоставить пользователю соответствующий тип данных. Давайте так и поступим.

Объявим регулярный тип «сети», все объекты которого устроены аналогично массиву «сеть»:

```
(а) type сети is array (узел) of  запись_об_узле;
```

Теперь объект «сеть» можно было бы объявить так:

```
(б) сеть: сети;
```

Так же может поступить и пользователь, если ему понадобятся, например, две сети:

```
(в) сеть1, сеть2: сети;
```

и к его услугам два объекта из нужного класса.

Но возникает несколько вопросов.

Во-первых, мы подчеркивали, что объект «сеть» недоступен пользователю непосредственно (это и гарантировало целостность). Точнее, имя этого объекта не было видимо пользователю. А чтобы писать объявления вида (в), пользователь должен явно выписать имя «сети». Другими словами, объявление типа «сети» должно быть видимым пользователю!

Но видимые извне пакета объявления должны находиться в его спецификации. Куда же именно в спецификации пакета следует поместить объявление (а)? Вспомним правило последовательного определения. В (а) использованы имена типов «узел» и запись_об_узле. Но последний пока не объявлен в спецификации нашего пакета. Значит, нужно объявить и этот тип (после типа «связи», то есть после строки 11), затем поместить (а).

Во-вторых, следует подправить определения и реализацию операций. Ведь пока они работают с одной-единственной сетью. Нужно сделать так, чтобы пользователь мог указать интересующую его сеть в качестве аргумента операции. К тому же грамотный программист всегда стремится сохранить преемственность со старой версией программы (гарантировать ранее написанным программам пользователя возможность работать без каких-либо изменений). Поэтому следует разрешить пользователю работать и по-старому, с одной и той же сетью, когда аргумент не задан, и по-новому, с разными сетями, когда аргумент задан.

Для этого перепишем строки с 13 по 18 спецификации пакета:

```
(13') procedure вставить (X : in узел, в_сеть : in out сети);
```

Обратите внимание, режим второго параметра – in out! Указанная им сеть служит обновляемым параметром (результатом работы процедуры «вставить» служит сеть со вставленным узлом).

```
(14') procedure удалить (X : in узел, из_сети : in out сети);
(15') procedure связать (A, B : in узел, в_сети : in out сети);
(17') function  узел_есть (X: узел, в_сети: сети) return BOOLEAN;
(18') function  все_связи (X : узел, в_сети : сети) return связи;
```

Так как нужно обеспечить и возможность работать по-старому, с одной сетью, то в спецификации пакета следует оставить строки и 13–18, и 13'–18'. Тогда в соответствии с правилами перекрытия операций в зависимости от заданного количества аргументов будет вызываться нужная спецификация (и, конечно, нужное тело) операции. Например, написав

```
удалить(33);
```

вызовем строку 14 (и соответствующее тело для этой процедуры), а написав

```
удалить(33, сеть1);
```

или лучше

```
удалить(33, из_сети => сеть1);
```

вызовем строку 14' (и еще не созданное нами тело для этой процедуры).

Однако пора вспомнить о принципиальной проблеме, которая, возможно, уже давно мучит вдумчивого читателя. Ведь теперь не гарантирована целостность сетей!

Для того мы и скрывали «сеть» в теле пакета, чтобы пользователь не мог написать, например,

```
сеть(33).связан.число:= 7;
```

и нарушить тем самым дисциплину работы с сетью так, что последующее выполнение процедуры

```
удалить(33);
```

(в которой есть цикл по массиву связей узла 33) может привести к непредсказуемым последствиям.

Введя объявление (в), пользователь может нарушить целостность объекта сеть1 оператором

```
сеть1(33).связан.число := 7;
```

Теперь читателю должна стать полностью понятной принципиальная важность концепции регламентированного доступа к объектам класса «сети», упомянутой при постановке задачи: нужно позволить пользователю создавать новые сети и работать с ними посредством объявленных операций, но нужно защитить сети от нежелательного (несанкционированного) доступа. Обратите внимание, раньше нежелательный доступ к объекту «сеть» был невозможен потому, что объект был невидим пользователю, скрыт в теле пакета. Объявив тип «сети» в спецификации, мы сделали видимыми для пользователя и имя типа, и имена (селекторы) полей объектов этого типа.

Итак, основное противоречие – в том, что скрыть объявление типа «сети» в теле пакета нельзя – ведь его нужно оставить видимым пользователю (чтобы он мог объявлять новые сети), а оставить это объявление полностью открытым также нельзя – пользователь может нарушить целостность сетей.

Вот если бы ввести тип так, чтобы его имя было видимо, а строение объектов – невидимо (то есть разделить его спецификацию и реализацию)! Тогда технологическая потребность в регламентированном доступе была бы полностью удовлетворена.

Эта красивая и естественная идея в Аде воплощена в концепции ПРИВАТНЫХ типов данных (в Модуле-2 – в концепции так называемых «непрозрачных» типов данных).

4.3.2. Приватные типы данных

Подумаем о выразительных средствах, реализующих концепцию регламентированного доступа к данным определенного типа.

По существу, нужно определить некоторую абстракцию данных – проявить то, что существенно (для пользователя – имя типа и операции с объектами этого типа), и скрыть то, что несущественно (и даже вредно знать пользователю – строение объектов). Но ведь аналогичная задача для операционных абстракций решается легко: в спецификацию пакета помещается то, что должно быть видимым (спецификация операции), а в тело – то, что следует скрыть (полное определение операции).

Так что было бы естественным аналогично оформить абстракцию данных – поместить в спецификацию пакета то, что должно быть видимым (что может играть роль минимальной «спецификации типа»), а в тело пакета упрятать полное определение типа.

В Аде минимальная «спецификация типа» воплощена конструктом «объявление приватного типа», например:

```
(a')   type сети is private;
```

а также перечнем спецификаций применимых операций.

Полное определение приватного типа по аналогии с определениями операций кажется естественным поместить в тело пакета. (Именно так сделано в Модуле-2.)

Почти так и нужно поступать в Аде. Но полное объявление приватного типа приходится помещать не в тело пакета, а в «полузакрытую» (ПРИВАТНУЮ) часть спецификации пакета, отделяемую от открытой части ключевым словом private.

Спецификация обновленного пакета, который назовем «управление_сетями», может выглядеть следующим образом:

```
package управление_сетями is
  ...   — как и раньше; строки 2-11.
  type сети is private;
  ... — операции над сетями
  ... — строки 13-18.
  ... — строки 13'-18'.
private
   type запись_об_узле is
   record
      включен : bollean := false;
      связан : связи;
   end record;
   type сети is array (узел) of запись_об_узле;
end управление_сетями;
```

В общем случае спецификация пакета имеет вид:

```
package имя_пакета is
   объявления_видимой_части
   [private
   объявления_приватной_части ]
end имя_пакета;
```

Квадратные скобки указывают, что приватной части может и не быть (как, например, в пакете управление_сетью).

Зачем же в Аде понадобилась приватная часть? Почему нет полной аналогии между операционными абстракциями и абстракцией данных? (В языке Модула-2 эта аналогия выдержана полностью.) На эти вопросы ответим позже.

Семантика приватной части проста. Эту часть можно считать «резидентом тела пакета» в его спецификации. В теле пакета непосредственно доступны все имена, объявленные в спецификации пакета, в том числе и в приватной части.

С другой стороны, в использующих этот пакет сегментах видимы только объявления открытой части спецификации.

Напишем один из таких использующих сегментов – процедуру две_сети:

```
with управление_сетями; use управление_сетями;
procedure две_сети is
 сеть1, сеть2 : сети;
begin
   вставить(13, в_сеть => сеть1 );
   вставить(33, в_сеть => сеть1 );
   связать(13, 33, в_сети => сеть1);
   сеть2 := сеть1; – присваивание полных объектов !
. . .
end две_сети;
```

Когда управление дойдет до места, отмеченного многоточием, будут созданы две сети: «сеть1» и «сеть2», с узлами 33 и 13, причем эти узлы окажутся связанными между собой.

Таким образом, пользователи могут создавать сети и работать с ними с полной гарантией целостности – все требуемые услуги предоставлены нашим обновленным пакетом. Конечно, для этого нужно дополнить тело пакета, поместив туда определения новых операций. Оставим это в качестве упражнения.

Вопрос. Нельзя ли испортить сеть за счет того, что доступен тип «связи»? Ведь становится известной структура связей указанного узла.

Подсказка. А где средства для несанкционированного изменения этой структуры?

Итак, концепция регламентированного доступа в Аде воплощена разделением спецификации и реализации услуг (разделением спецификации и тела пакета), а также приватными типами данных.

Доступ к «приватным» данным возможен лишь посредством операций, объявленных в ОПРЕДЕЛЯЮЩЕМ ПАКЕТЕ. Для любого определяемого типа данных (не только приватного) так называется пакет, где расположено объявление этого типа данных (для типов «сети», «узел», «связи» и др. таковым служит пакет управление_сетями). Невозможен доступ, основанный на знании строения объектов (то есть выборкой или индексацией), – это строение скрыто в приватной части и в использующих сегментах неизвестно.

Подчеркнем, что в теле определяющего пакета объекты приватных типов ничем не отличаются от любых других – их строение известно, выборка и индексация разрешены!

4.3.3. Строго регламентированный доступ. Ограниченные приватные типы

В общем случае к объектам приватных типов применимы также операции присваивания и сравнения на равенство и неравенство (полных объектов, как в процедуре две_сети!). Хотя это и удобно (такие операции часто нужны, и неразумно за-

ставлять программистов определять их для каждого приватного типа), все-таки концепция строго регламентированного доступа в таких типах не выдержана до конца. В Аде она точно воплощена лишь в так называемых ОГРАНИЧЕННЫХ приватных типах. К объектам таких типов неприменимы никакие предопределенные операции, в том числе присваивания и сравнения, – все нужно явно определять (в определяющем пакете). Объявления ограниченных приватных типов выделяются ключевыми словами limited private, например:

```
type   ключ  is  limited private;
```

Применение к объектам типа «ключ» операций присваивания или сравнения вызовет сообщение об ошибке, если только в определяющем пакете для этого типа не определены свои собственные операции, обозначаемые через «:=», «=» или «/=».

Так как ограниченные приватные типы точно воплощают идею строго регламентированного доступа, интересно понять, в каких же задачах столь строгие ограничения на присваивания и сравнения могут быть существенными.

Присваивания. Мы уже упоминали, что бывают ЯП вовсе без присваивания. Таковы чистый Лисп, Базисный Рефал, функциональные и реляционные языки. Подробнее поговорим о них позже. Чтобы соответствовать роли базового языка, Ада должна предоставлять средства развития, позволяющие моделировать и такие ЯП, причем моделировать в точности, с гарантией защиты создаваемых абстракций. Так что ограниченные приватные типы вместе со средством их определения (пакетом) – важнейшее средство развития в современном базовом ЯП.

> **Упражнение** (повышенной сложности). Создайте определяющий пакет для каждой из рассмотренных далее моделей ЯП.
>
> **Замечание** (о наблюдении Дейкстры). Подтверждением принципа цельности (согласованности данных, операций, связывания) служит наблюдение Дейкстры, что присваивание нужно лишь в тех языках, где есть повторения (циклы). Именно в циклах появляются переменные, которым нужно присваивать значения (постоянные относительно очередного исполнения тела цикла).
> Когда циклов нет и потенциальная бесконечность обслуживается рекурсией (в Лиспе, Рефале и т. п.), достаточна ИНИЦИАЛИЗАЦИЯ (частный случай присваивания – присваивание начальных значений). Так что наличие циклов должно быть согласовано не только с другими операциями (присваиванием), но и с данными (появляются переменные), а также со связыванием (областью действия каждой переменной в таких ЯП в идеале должен быть некоторый цикл).
>
> **Вопрос.** Что можно сказать в этих условиях об исходных данных и результатах?

Рассмотрим пример. Допустим, что нужно моделировать выпуск изделий с уникальными номерами (вспомним заводские номера двигателей, автомобилей, самолетов и т. п.). Естественно считать, что объект приватного типа «изделие» – результат базовой для этого типа функции «создать», генерирующей очередное «изделие». Если позволить присваивать объекты этого типа переменным, то ста-

нет возможным их дублировать, и уникальность будет нарушена. Поэтому тип «изделие» должен быть ограниченным приватным.

Сравнения. Во-первых, операции сравнения для объектов некоторых типов могут попросту не иметь смысла, то есть объекты могут быть несравнимыми. Из соображений надежности попытку их сравнения следует квалифицировать как ошибку (то есть сравнение должно быть запрещено). Так, при первоначальном знакомстве с Адой упоминались задачные типы, объекты которых – параллельно исполняемые задачи. Сравнение таких объектов на равенство (при отсутствии присваивания) бессмысленно просто потому, что любой из них уникален по определению. Да и содержательно трудно приписать какой-либо естественный смысл сравнению на равенство (а не, например, на подобие) двух независимо исполняемых задач. Ведь они в постоянном изменении, причем асинхронном, а всякое реальное сравнение занимает время.

Поэтому в Аде задачные типы – ограниченные приватные по определению, и, следовательно, всякая попытка сравнить задачи на равенство квалифицируется как ошибка. Кстати, любые составные типы с компонентами ограниченного типа считаются ограниченными.

Во-вторых, нежелание допускать сравнение на равенство может быть связано с защитой от нежелательного доступа, с обеспечением секретности. Так, и пользователи, и файлы в файловой системе могут быть снабжены атрибутами типа «ключ». Однако неразумно разрешать пользователю сравнивать ключи, чтобы решать, пользоваться файлом или нет. Право сравнивать ключи и разрешать доступ должно быть только у самой файловой системы. Поэтому для пользователя тип «ключ» должен быть ограниченным – он может его передать другому, но не может «подделать» или «подобрать».

4.3.4. Инкапсуляция

Определение приватных типов данных – один из примеров **инкапсуляции** – заключения в «защитную оболочку», предохраняющую от разрушения. Мы видели, как недоступность (защита) строения объектов приватных типов от произвольного доступа со стороны пользователя гарантирует их целостность в некотором содержательном смысле. Понятие инкапсуляции по общности занимает промежуточное положение между концепцией регламентированного доступа и ее конкретной реализацией приватными типами данных.

Приватные типы Ады с точки зрения пользователя – это инкапсулированные (защищенные) типы данных. Однако с точки зрения реализатора тела определяющего пакета – это обычные незащищенные типы. В общем случае инкапсулировать можно не только типы, но и отдельные объекты или системы объектов (таковы объекты, объявленные в теле пакета).

Итак, задача о моделировании сетей помогла проявить технологическую потребность в инкапсуляции и дала повод поработать с конструктами, удовлетворяющими эту потребность, – с объявлениями приватного типа и пакетами.

4.4. Характеристики, связанные с типом. Класс значений, базовый набор операций

Продолжим анализ общей концепции типа данных. До сих пор мы концентрировали внимание только на способе привязки характеристик к объектам данных. Сутью характеристик мы при этом не интересовались. Перед нами очередной пример разумной абстракции. Абстракция от сути характеристик привела к концепции уникальности типа. Концепция уникальности – к простоте прогнозирования-контроля. А простота прогнозирования-контроля позволяет по-новому взглянуть на саму концепцию типа.

Действительно, в ранних ЯП под характеристиками данных обычно понимались характеристики их значений (уже упоминавшиеся разрядность, точность, вид выделяемой памяти и т. п.). Конечно, неявно всегда подразумевалось, что к значениям с одними характеристиками применимы одни операции, а к значениям с другими характеристиками – другие. И раньше чувствовалось, что класс применимых операций – существенная содержательная характеристика класса данных (ведь это естественно следует из понятия данного как разумной абстракции от конкретной операции).

Но идею явного связывания класса данных с классом применимых операций трудно воплотить в рамках структурной эквивалентности типов. Ведь для определения класса данных, к которым применима операция, придется (как, например, в Алголе-68) производить нетривиальные расчеты совокупности данных, структурно эквивалентных параметрам операции.

Именная эквивалентность в концепции уникальности упростила связывание с типом любых характеристик. Легко узнать и классы данных, связанных с любой операцией, – имена типов явно указаны в спецификациях ее параметров.

Может показаться, что нетрудно проделать и обратное – указать для каждого типа все применимые операции. Но представим себе, что пакет управление_сетями предоставлен в распоряжение большого коллектива пользователей. Каждый из них волен написать сегменты, использующие этот пакет, и в этих сегментах ввести свои операции над объектами типа «сети». Например, один ввел процедуру выделения связных подсетей, другой – процедуру подсчета хроматического числа сети, третий – вычисления минимального маршрута между двумя узлами. Следует ли считать характеристикой типа «сети» набор из всех этих операций? И каков содержательный смысл в такой характеристике? Ведь в разных контекстах доступны разные элементы этого набора.

Лучше считать характеристикой типа относительно постоянное его свойство, связанное с ним в любых контекстах. Таковы КЛАСС ЗНАЧЕНИЙ объектов этого типа и БАЗОВЫЙ НАБОР ОПЕРАЦИЙ, применимых к этим объектам. В Аде эти две характеристики связываются с типом соответственно ОБЪЯВЛЕНИЕМ ТИПА и ОПРЕДЕЛЯЮЩИМ ПАКЕТОМ.

4.5. Воплощение концепции уникальности типа. Определение и использование типа в Аде (начало)

Концепция типа воплощена в Аде в основном четырьмя конструктами: объявлением типа, пакетом, объявлением подтипа и объявлением объекта. В совокупности они и позволяют считать, что тип в Аде обладает, кроме имени, еще двумя важнейшими характеристиками – классом значений и набором применимых операций. С каждым из названных четырех конструктов мы уже встречались. Поговорим подробнее об их строении, смысле и взаимодействии.

4.5.1. Объявление типа. Конструктор типа. Определяющий пакет

Объявление типа вводит имя нового типа и связывает это имя с конструктором типа. Последний служит для создания нового типа из уже известных и располагается в объявлениях типов после ключевого слова is.

Создать (объявить) новый тип в Аде – значит определить класс допустимых значений объектов этого типа и набор базовых операций, связанных с создаваемым типом.

В Аде предопределены некоторые типы данных (обслуживающие наиболее общие потребности проблемной области – универсальный_целый, универсальный_вещественный и др.), а также класс допустимых значений перечисляемых типов. Кроме того, предопределены операции, применимые к объектам целых категорий типов.

Например, со всеми регулярными типами связана операция получения указателя на компоненту массива (индексация), со всеми комбинированными типами связана операция получения указателя на компоненту записи (выборка), со всеми так называемыми ДИСКРЕТНЫМИ типами – получение по заданному значению последующего или предыдущего значения. Если явно не оговорено обратное, то к объектам любого типа можно применять сравнение на равенство и неравенство, извлечение и присваивание значения.

Предопределенные типы, классы значений и операции служат исходным материалом для нескольких категорий конструкторов типа – у каждой категории типов свой конструктор.

Объявление типа (а следовательно, и конструктор типа) служит для того, чтобы полностью и окончательно определить класс допустимых значений и (в общем случае лишь частично) определить базовые операции.

Полный и окончательный набор базовых операций некоторого типа фиксируется его определяющим пакетом (так называется пакет, спецификация которого содержит объявление этого типа). В спецификации определяющего пакета вместе

с объявлением нового типа могут присутствовать и объявления новых операций, связанных с ним.

Например, конструктор производного типа, создавая тип «узел», определяет для него класс (в данном случае – диапазон) значений и связывает с типом «узел» обычные операции над целыми числами (наследуемые у родительского типа INTEGER). Остальные базовые операции для объектов типа «узел» (вставить, удалить, связать и др.) определены в конце спецификации пакета управление_сетью, который и служит для этого типа определяющим пакетом. Подчеркнем, что для типов, объявляемых в теле пакета, он, естественно, определяющим не считается (**почему?**).

Еще пример. Конструктор КОМБИНИРОВАННОГО типа, создавая тип «связи», определяет для него, во-первых, класс значений (записи с двумя полями – первое с СЕЛЕКТОРОМ «число» типа число_связей, второе с селектором «узлы» типа «перечень-связей»). Во-вторых, этот конструктор определяет обычные для всех комбинированных типов операции доступа как ко всему значению объекта (по имени объекта), так и к его компонентам (по составным именам с использованием селекторов). Еще одна (и последняя) базовая операция для этого типа определена в пакете управление_сетью – это операция все_связи. Доступ к полному значению объекта типа «связи» использован в реализации функции все_связи (в операторе возврата), а доступ к одному полю по селектору – в реализации операций вставить, удалить, чистить, переписать.

Конструктор ПРИВАТНОГО типа, создавая тип «сети», определяет в качестве базовых операций только присваивание и сравнение на равенство и неравенство. Определяющий пакет управление_сетями добавляет базовые операции вставить, удалить и др.

> Обратите внимание, одни и те же операции могут быть базовыми для различных типов! За счет чего?

Класс значений для типа «сети» определен полным объявлением в приватной части, однако пользователю этот класс остается «неизвестным» (непосредственно недоступным).

Запас предопределенных типов, значений и операций – это базис ЯП, а конструкторы типа – характерный пример средств развития. Процесс развития ЯП (с помощью любых конструкторов) начинается с применения конструкторов к базису. На очередном шаге развития конструкторы применяются к любым уже определенным сущностям (в том числе и к базисным).

4.6. Конкретные категории типов

4.6.1. Перечисляемые типы. «Морская задача»

Займемся технологической потребностью, которая часто встречается в так называемых дискретных задачах (в дискретной математике вообще). Речь идет о потребности работать с относительно небольшими конечными множествами.

Замечание (о конечных множествах). Когда конечные множества очень большие, то с точки зрения программирования они могут оказаться неотличимыми от множеств бесконечных. Специфика конечного сказывается тогда, когда мощность множества удается явно учесть в программе. При этом в соответствии с принципом согласования абстракций средства объявления конечных множеств должны быть согласованы со средствами манипулирования как множествами в целом, так и отдельными их элементами.

Рассмотрим очень упрощенную задачу управления маневрами корабля.

Содержательная постановка. Корабль движется по определенному курсу. Поступает приказ изменить курс. Требуется рассчитать команду, которую следует отдать рулевому, чтобы направить корабль по новому курсу (совершить нужный маневр).

Модель задачи. Будем считать, что возможных курсов только четыре: север, восток, юг и запад. Можно отдать лишь одну из четырех команд: прямо, налево, направо и назад. Исполнение команд «налево» и «направо» изменяет курс на 90 градусов. Требуется рассчитать новую команду по заданным старому и новому курсам. Например, для старого курса «восток» и нового курса «север» нужный маневр выполняется по команде «налево».

Программирование (полная формализация задачи). Попытаемся вновь применить пошаговую детализацию.

Шаг 1. Нужно «рассчитать» команду. Это уже не комплекс услуг, а одна операция. Представим ее функцией, для которой старый и новый курсы служат аргументами, а команда рулевому – результатом. Чтобы сразу написать спецификацию такой функции на Аде, нужно верить, что на последующих шагах можно будет ввести подходящие типы аргументов и результата. Поверим, не думая пока об особенностях этих типов. Тогда нужную операционную абстракцию можно воплотить следующей спецификацией:

```
function маневр(старый, новый : курс) return команда;
```

Шаг 2. Как формализовать понятие «команда»?

С точки зрения ЯП, это должен быть тип данных – ведь имя «команда» использовано в спецификации параметров функции. Следовательно, нужно определить связанные с этим типом значения и базовые операции.

С содержательной точки зрения ясно, что нужны только названия команд. Что с ними можно делать – не ясно, это выходит за рамки модели нашей задачи (наша модель неполна в том отношении, что в ней отсутствуют, например, модель устройства корабля и модель рулевого как части корабля). Поэтому, формализуя понятие «команда», удовлетворимся тем, что проявим список существенных команд и отразим в названиях команд их роль в управлении кораблем.

Существенных команд всего четыре: прямо, налево, направо, назад. Таким образом, нужно объявить тип данных с четырьмя перечисленными значениями. Ада позволяет это сделать с помощью следующего объявления ПЕРЕЧИСЛЯЕМОГО типа:

```
type команда is(прямо, налево, направо, назад);
```

Не зря эта категория типов названа «перечисляемыми типами» – все допустимые значения явно перечислены.

Такие типы называют иногда «перечислимыми». Однако этот термин в математике занят, относится тоже к множествам и имеет другой смысл (перечислимые множества вполне могут быть бесконечными). К тому же специфика рассматриваемых типов в том и состоит, что их значения явно перечисляют при определении такого типа.

Перечисляемые типы придуманы Н. Виртом и впервые появились в созданном им языке Паскаль. В наших примерах до сих пор были такие типы, для которых имело смысл говорить об описании множества значений, о создании нового или разрушении старого значения. Но никогда не шла речь о явном перечислении в программе всех значений некоторого типа. Другими словами, определения типов были всегда интенсиональными и никогда – экстенсиональными. Это было и не нужно, и, как правило, невозможно – шла ли речь о целых или вещественных числах, о массивах или записях. «Не нужно» означает, что такова была дисциплина применения этих типов. «Невозможно» связано с их практической бесконечностью.

В нашем случае и можно, и нужно давать экстенсиональное определение типа, явно перечислив все значения типа «команда». Можно, потому что их всего четыре, и мы их уже перечислили. А зачем это нужно?

Нужно потому, что всякое интенсиональное определение опирается на существенные индивидуальные свойства элементов определяемого множества. А в нашем случае таких свойств нет! Годятся любые различные имена для команд (желательно мнемоничные, отражающие содержательную роль команд в нашей задаче). Захотим – будет {прямо, налево, направо, назад}, не понравится – сделаем {вперед, влево, вправо, обратно} или {так_держать, лево_руля, право_руля, задний_ход} и т. п. Поэтому нет иного способа догадаться о принадлежности конкретного имени к типу «команда», кроме как увидеть его в соответствующем списке.

Итак, объявление перечисляемого типа вводит мнемонические имена для компонент модели решаемой задачи (в нашем случае – имена команд). До изобретения Вирта программисты явно кодировали такие компоненты (обычно целыми числами). В сущности, Вирт предложил полезную абстракцию от конкретной кодировки, резко повышающую надежность, понятность и модифицируемость программ без заметной потери эффективности.

Упражнение. Обоснуйте последнее утверждение.

Подсказка. См. ниже стр. 113.

Шаг 3. Как формализовать понятие «курс»?

Нетрудно догадаться, что «курс» должен быть перечисляемым типом:

```
type курс is(север, восток, юг, запад);
```

Ведь снова, как и для типа «команда», с точки зрения решаемой задачи абсолютно несущественно внутреннее строение этих значений. Поэтому невозможно вводить «курс» каким-либо конструктором составного типа. Действительно, из

чего состоит «север» или «восток»? Важно лишь, что это разные сущности, связанные между собой только тем, что направо от севера – восток, а налево от востока – север.

Таким образом, значения типа «курс», так же как и типа «команда», следует считать просто именами компонент модели задачи (точнее, той модели внешнего мира, на которой мы решаем нашу содержательную задачу). Существенные связи этих имен – непосредственное отражение содержательных связей между именуемыми компонентами внешней модели. Мы пришли еще к одной причине, по которой нам нужны в программе все такие имена-значения, – иначе не запрограммировать базовые функции (ведь нет никаких внутренних зависимостей между значениями, только внешние, а их-то и нужно отражать).

Когда мы программировали базовую функцию «связать» в пакете управление_сетью, нам, наоборот, были абсолютно безразличны индивидуальные имена узлов – можно было программировать, опираясь на внутреннее строение именуемых объектов (на строение записей_об_узле в массиве «сеть»). Это внутреннее строение создавалось пользователем, работающим с пакетом, посредством других базовых операций.

Когда же мы будем программировать, например, операцию «налево», никакое внутреннее строение не подскажет нам, что налево от юга находится восток. Это следует только из модели внешнего мира, создаваемой нами самими, то есть в данном случае создателем пакета, а не пользователем. Поэтому мы обязаны явно сопоставить восток – югу, север – востоку, юг – западу.

> С ситуацией, когда строение значений скрыто от пользователя, но отнюдь не безразлично для реализатора, мы уже встречались, когда изучали приватные типы данных. Теперь строение не скрыто, но существенно лишь постольку, поскольку обеспечивает идентификацию значений. В остальном можно считать, что его просто нет, – перед нами список имен объектов внешнего мира, играющих определенные роли в решаемой задаче.
> Перечисляемые типы похожи на приватные тем, что также создают для пользователя определенный уровень абстракции – пользователь вынужден работать только посредством операций, явно введенных для этих типов. Однако если операции приватных типов обычно обеспечивают доступ к скрытым компонентам содержательных «приватных» объектов, то операции перечисляемых типов отражают связи между теми содержательными объектами, которые названы явно перечисленными в объявлении типа именами. Вместе с тем приватный тип вполне может оказаться реализованным некоторым перечисляемым типом.

Завершая шаг детализации, определим перечисляемый тип «курс» вместе с базовыми операциями-поворотами. Сделаем это с помощью спецификации пакета «движение»:

```
package движение is
   type курс is(север, восток, юг, запад);
   function налево(старый : курс) return курс;
   function направо(старый : курс) return курс;
   function назад(старый : курс) return курс;
end движение;
```

Шаг 4. Тело функции «маневр».

Идея в том, чтобы понять, каким поворотом можно добиться движения в нужном направлении, и выдать соответствующую команду:

```
function маневр(старый, новый : курс) return команда;
begin
   if новый = старый then return прямо;
   elsif новый = налево(старый) then return налево;
   elsif новый = направо(старый) then return направо;
   else return назад;
   end if;
end маневр;
```

Мы свободно пользовались сравнением имен-значений на равенство.

Условный оператор с ключевым словом elsif можно считать сокращением обычного условного оператора. Например, оператор

```
if B1 then S1;
   elsif B2 then S2;
   elsif B3 then S3;
end if;
```

эквивалентен оператору

```
if B1 then S1
else
   if B2 then S2
   else
      if B3 then S3
      end if;
   end if;
end if;
```

Такое сокращение удобно, когда нужно проверять несколько условий последовательно.

Программирование функции «маневр» завершено.

Если считать, что «маневр» – лишь одна из предоставляемых пользователю услуг, можно включить ее в пакет со следующей спецификацией:

```
package услуги is
   type команда is(прямо, налево, направо, назад);
   package движение is
      type курс is(север, восток, юг, запад);
      function налево(старый : курс) return курс;
      function направо(старый : курс) return курс;
      function назад(старый : курс) return курс;
   end движение;

   use движение;
   function маневр(старый, новый : курс) return команда;
end услуги;
```

Замечания о конструктах. Во-первых, видно, что пакет («движение») может быть объявлен в другом пакете. Чтобы воспользоваться объявленными во внутреннем

пакете именами при объявлении функции «маневр», указание контекста (with) не нужно. Но указание сокращений (use) обязательно, если желательно пользоваться сокращенными именами. Иначе пришлось бы писать

```
function маневр(старый,новый : движение.курс) return команда;
```

Во-вторых, имена функций совпадают с именами команд (обратите внимание на тело функции «маневр»). Это допустимо. Даже если бы возникла коллизия наименований, имена функций всегда можно употребить с префиксом – именем пакета. Например, движение.налево, движение.назад, а имена команд – употребить с так называемым КВАЛИФИКАТОРОМ. Например, команда (налево), команда (направо), команда (назад). На самом деле в нашем случае ни префиксы, ни квалификаторы не нужны, так как успешно действуют правила перекрытия – по контексту понятно, где имена команд, а где – функции.
В-третьих, функция «маневр» применима к типу данных «курс», но не входит в набор его базовых операций. Ведь определяющий пакет для этого типа – «движение». Зато для типа «команда» функция «маневр» – базовая операция.

Шаг 5. Функции пакета «движение»

```
function налево(старый : курс) return курс is
begin
   case старый of
      when север   => return запад;
      when восток  => return север;
      when юг      => return восток;
      when запад   => return юг;
      end case;
end налево;
```

Замечание (о согласовании абстракций). Перед нами – наглядное подтверждение принципа цельности. Раз в Аде есть способ явно описывать «малые» множества (вводить перечисляемые типы), то должно быть и средство, позволяющее непосредственно сопоставить определенное действие с каждым элементом множества. Таким средством и служит ВЫБИРАЮЩИЙ ОПЕРАТОР (case). Между ключевыми словами case и of записывается УПРАВЛЯЮЩЕЕ ВЫРАЖЕНИЕ некоторого перечисляемого типа (точнее, любого ДИСКРЕТНОГО типа – к последним относятся и перечисляемые, и целые типы с ограниченным диапазоном значений). Между of и end case записываются так называемые ВАРИАНТЫ. Непосредственно после when (когда) записывается одно значение, несколько значений или диапазон значений указанного типа, а после «=>» – последовательность операторов, которую нужно выполнить тогда и только тогда, когда значение управляющего выражения равно указанному значению (или попадает в указанный диапазон).

Выбирающий оператор заменяет условный оператор вида

```
if старый = север     then return запад;
elsif старый = восток then return север;
elsif старый = юг     then return восток;
elsif старый = запад  then return юг ;
end if;
```

Условный и выбирающий операторы – частные случаи развилки (одной из трех основных управляющих структур: последовательность, развилка, цикл, – используемых в структурном программировании).

По сравнению с условным, выбирающий оператор, во-первых, компактнее (не нужно повторять выражение); во-вторых, надежнее – и это главное.

Дело в том, что варианты обязаны охватывать все допустимые значения анализируемого типа и никакое значение не должно соответствовать двум вариантам. Все задаваемые после when значения (и диапазоны) должны быть вычисляемы статически (то есть не должны зависеть от исходных данных программы, с тем чтобы компилятор мог их вычислить). Так что компилятор в состоянии проверить указанные требования к вариантам выбирающего оператора и обнаружить ошибки.

Наконец, статическая вычислимость обеспечивает и третье преимущество выбирающего оператора – его можно эффективно реализовать (значение выбирающего выражения может служить смещением относительно начала вариантов).

Предоставим возможность читателю самостоятельно запрограммировать функции «направо» и «назад», завершив тем самым решение нашей «морской» задачи.

Морская задача и Алгол-60. Читателям, привыкшим к Паскалю, где имеются перечисляемые типы и выбирающий оператор, не так просто оценить достижение Вирта. Чтобы подчеркнуть его связь с перспективной технологией программирования (и заодно лишний раз подтвердить принцип технологичности), посмотрим на «морскую» задачу из другой языковой среды. Представим, что в нашем распоряжении не Ада, а Алгол-60.

Технология. Уже на первом шаге детализации нам не удалось бы ввести подходящую операционную абстракцию. Помните, нам была нужна уверенность в возможности определить подходящие типы для понятий «курс» и «команда». В Алголе-60 вообще нет возможности определять типы, в частности перечисляемые. Поэтому пришлось бы «закодировать» курсы и команды целыми числами. Скажем, север – 1, восток – 2, юг – 3, запад – 4; команда «прямо» – 1, «налево» – 2, «направо» – 3, «назад» – 4. Заголовок функции «маневр» выглядел бы, например, так:

```
integer procedure маневр(старый, новый);
integer старый, новый; value старый, новый;
```

Приступая к проектированию тела функции, мы не имели бы случая предварительно создать абстрактный тип «курс» с операциями поворота. Но ведь именно с операциями поворота связана основная идея реализации функции «маневр» на Аде! Вспомните, чтобы подобрать подходящую команду, мы проверяли возможность получить новый курс из старого определенным поворотом. Если бы применяемая технология программирования не требовала вводить абстрактный тип «курс», то и идея реализации функции «маневр» вполне могла оказаться совсем другой. Не было бы удивительным, если бы ее тело было запрограммировано «в лоб», например так:

```
begin маневр :=
   if старый = новый then 1
   else if старый = 1 & новый = 4 v старый = 2 & новый = 1 v
      старый = 3 & новый = 2 v старый = 4 & новый = 3 then 2
   else if старый = 1 & новый = 2 v старый = 2 & новый = 3 v
      старый = 3 & новый = 4 v старый = 4 & новый = 1 then 3
   else if старый = 1 & новый = 3 v старый = 2 & новый = 4 v
      старый = 3 & новый = 1 v старый = 4 & новый = 2 then 4;
end маневр;
```

Конечно, «настоящие» программисты постарались бы «подогнать» кодировку курсов и команд, с тем чтобы заменить прямой перебор «вычислением» команды. Однако такой прием неустойчив по отношению к изменению условий задачи, и в общем случае решение с большой вероятностью может оказаться ошибочным. К тому же оно менее понятно по сравнению с решением на Аде. Нужно «держать в голове» кодировку, чтобы понимать программу.

Таким образом, отсутствие средств, поддерживающих нужные абстракции (в частности, в процессе пошаговой детализации), вполне может помешать и наиболее творческим моментам в программировании, помешать увидеть изящное, надежное, понятное и эффективное решение.

Надежность. Внимательнее сравним программы на Аде и Алголе-60 с точки зрения надежности предоставляемой услуги. Чтобы воспользоваться операцией «маневр», на Аде можно написать, например,

```
маневр(север, восток);
```

а на Алголе-60

```
маневр(1,2);
```

Ясно, что первое – нагляднее, понятнее (а значит, и надежнее). Но высокий уровень надежности гарантируется не только наглядностью, но и контролем при трансляции. На Аде нельзя написать маневр (1,2), так как транслятор обнаружит несоответствие типов аргументов и параметров! А на Алголе-60 можно написать

```
маневр(25,30);
```

и получить... неизвестно что.

А чтобы получить тот же уровень контроля, который автоматически обеспечивает Ада-компилятор, нужно добавить в программу явные проверки диапазона целых значений и обращение к соответствующим диагностическим процедурам. И все это будет работать динамически, а в Аде – статически. Так что надежность при программировании на Алголе-60 может быть обеспечена усилиями только самого программиста и только за счет снижения эффективности целевой программы.

Можно постараться добиться большей наглядности, введя переменные «север», «восток», «юг» и «запад» (постоянных в Алголе-60 нет). Им придется присвоить значения 1, 2, 3, 4 также во время работы объектной программы, но зато окажется возможным писать столь же понятно, как и на Аде:

```
маневр(север, восток);
```

Однако в отличие от имен-значений перечисляемого типа в Аде, которые по определению – константы, эти переменные не защищены от случайных присваиваний. Не говоря уже о защите от применения к ним других операций (в Аде к значениям определенного перечисляемого типа применимы, конечно, только операции, параметры которых соответственно специфицированы).

Итак, мы выделили технологическую потребность определять небольшие множества имен и работать с ними на таком уровне абстракции, когда указываются лишь связи этих имен между собой и с другими программными объектами. Эта потребность и удовлетворяется в Аде перечисляемыми типами данных. Важно, что удовлетворяется она комплексно, в соответствии с важнейшими общими принципами (такими, как принцип цельности) и специфическими требованиями к ЯП (надежность, понятность и эффективность программ). Показано также, что перечисляемые типы не могут быть полностью заменены аппаратом классических ЯП.

4.6.2. Дискретные типы

Перечисляемые типы – частный случай так называемых ДИСКРЕТНЫХ типов.

Дискретным называется тип, класс значений которого образует ДИСКРЕТНЫЙ ДИАПАЗОН, то есть конечное линейно упорядоченное множество. Это значит, что в базовый набор операций для дискретных типов входит, во-первых, операция сравнения «меньше», обозначаемая обычно через «<»; во-вторых, функции «первый» и «последний», вырабатывающие в качестве результатов соответственно минимальный и максимальный элементы диапазона, и, в-третьих, функции «предыдущий» и «последующий» с очевидным смыслом. Эти операции для всех дискретных типов предопределены в языке Ада.

Кроме перечисляемых, дискретными в Аде являются еще и ЦЕЛЫЕ типы. Класс значений любого целого типа считается конечным. Для предопределенного типа INTEGER он фиксируется реализацией языка (то есть различные компиляторы могут обеспечивать различный диапазон предопределенных целых; этот диапазон должен быть указан в документации на компилятор; кроме того, его границы доставляются (АТРИБУТНЫМИ) функциями «первый» и «последний»). Для определяемых целых типов границы диапазона значений явно указываются в объявлении целого типа (см. объявление типа узел).

Для типа INTEGER предопределены также унарные операции «+», «–», «abs» и бинарные «+», «–», «*», «/», «**» (возведение в степень) и др.

Любые дискретные типы можно использовать для индексации и управления циклами. Мы уже встречались и с тем, и с другим в пакете управление_сетями.

В «морской» задаче мы не воспользовались предопределенными базовыми операциями для типа «курс». Но в соответствии с его объявлением

```
север < восток < юг < запад
```

причем

```
последующий(север) = восток;
предыдущий(восток) = север;
курс'первый = север;
```

```
курс'последний = запад;
```

Так что функцию «налево» можно было реализовать и так:

```
function налево(старый: курс) return курс is
begin
   case старый of
      when север => return запад;
      when others => return предыдущий(старый);
   end case;
end налево;
```

Обратите внимание, функция «предыдущий» не применима к первому элементу диапазона (как и функция «последующий» – к последнему элементу).

Вариант when others в выбирающем операторе работает тогда, когда значение выбирающего выражения (в нашем случае – значение параметра «старый») не соответствует никакому другому варианту. Выбирающее выражение должно быть дискретного типа (вот еще одно применение дискретных типов, кроме индексации и управления циклами), и каждое допустимое значение должно соответствовать одному и только одному варианту. Такое жесткое правило было бы очень обременительным без оборота when others. С его помощью можно использовать выбирающий оператор и тогда, когда границы диапазона изменяются при выполнении программы, то есть являются динамическими. Конечно, при этом изменяется не тип выбирающего выражения, а лишь его подтип – динамические границы не могут выходить за рамки статических границ, определяемых типом выражения.

Вообще, если D – некоторый дискретный тип, то справедливы следующие соотношения. Пусть X и Y – некоторые значения типа D. Тогда

```
последующий(предыдущий (X)) = X, если X /= D'первый;
предыдущий(последующий (X)) = X, если X /= D'последний;
          предыдущий (X) < X, если X /= D'первый.
```

Для дискретных типов предопределены также операции «<=», «>», «>=», «=», «/=».

Вот еще несколько примеров дискретных типов. Предопределены дискретные типы BOOLEAN, CHARACTER. При этом считается, что тип BOOLEAN введен объявлением вида

```
type BOOLEAN is(true, false);
```

так что true < false.

Для типа CHARACTER в определении языка явно перечислены 128 значений-символов, соответствующих стандартному коду ASCII, среди которых первые 32 – управляющие телеграфные символы, вторые 32 – это пробел, за которым следуют !»#$%&'()*+,–./0123456789:;<=>?, третьи 32 – это коммерческое at (@), за которым идут прописные латинские буквы, затем [\ ] ^_; наконец, последние 32 – знак ударения "; затем строчные латинские буквы, затем {|}, затем тильда ~ и символ вычеркивания. Для типов BOOLEAN, CHARACTER и INTEGER предопределены обычные операции для дискретных типов. (Мы привели не все такие опе-

рации.) Кроме того, для типа BOOLEAN предопределены обычные логические операции and, or, xor и not с обычным смыслом (xor – исключительное «или»).

Вот несколько примеров определяемых дискретных типов:

```
type день_недели is(пн, вт, ср, чт, пт, сб, вс);
type месяц is(январь, февраль, март, апрель, май, июнь, июль, август, сентябрь,
октябрь, ноябрь, декабрь);
type год is new INTEGER range 0..2099;
type этаж is new INTEGER range 1..100;
```

4.6.3. Ограничения и подтипы

Проанализируем еще одну технологическую потребность – потребность ограничивать множество значений объектов по сравнению с полным классом значений соответствующего типа. Рассмотрим фрагмент программы, меняющей знак каждого из десяти элементов вектора A:

```
for j in 1..10 loop
   A(j) :=-A(j);
end loop;
```

Перед нами фрагмент, который будет правильно работать только при условии, что вектор A состоит ровно из десяти элементов. Иначе либо некоторые элементы останутся со старыми знаками, либо индекс выйдет за границу массива. К тому же такой фрагмент способен работать только с вектором A и не применим к вектору B.

Другими словами, это очень конкретный фрагмент, приспособленный для работы только в специфическом контексте, плохо защищенный от некорректного использования.

Вопрос. В чем это проявляется?

Допустим, что потребность менять знак у элементов вектора возникает достаточно часто. Вместо того чтобы каждый раз писать аналогичные фрагменты, хотелось бы воспользоваться принципом обозначения повторяющегося и ввести подходящую подпрограмму, надежную и пригодную для работы с любыми векторами. Другими словами, мы хотим обозначить нечто общее, характерное для многих конкретных действий, то есть ввести операционную абстракцию.

От чего хотелось бы отвлечься? По-видимому, и от конкретного имени вектора, и от конкретной его длины. Возможно, от конкретного порядка обработки компонент или от конкретной размерности массива. Можно ли это сделать и целесообразно ли, зависит от многих причин. Но прежде всего – от возможностей применяемого ЯП. Точнее, от возможностей встроенного в него аппарата развития (аппарата абстракции-конкретизации).

Здесь уместно сформулировать весьма общий принцип проектирования (в частности, языкотворчества и программирования). Будем называть его принципом реальности абстракций.

Принцип реальности абстракций. Назовем **реальной** такую абстракцию, которая пригодна для конкретизации в используемой программной среде. Тогда принцип реальности абстракций можно сформулировать так:

в программировании непосредственно применимы лишь реальные абстракции.

Иначе говоря, создавая абстракцию, не забудь о конкретизации. Следует создавать возможность абстрагироваться только от таких характеристик, которые применяемый (создаваемый) аппарат конкретизации позволяет указывать в качестве параметров настройки.

В нашем примере попытаемся построить ряд все более мощных абстракций, следуя за особенностями средств развития в Аде. Скажем сразу: Ада позволяет явно построить первую из намеченных четырех абстракций (мы ведь собрались отвлечься от имени, от длины, от порядка и от размерности); со второй придется потрудиться (здесь-то и понадобятся ограничения и подтипы); третья потребует задачного типа и может быть построена лишь частично, а четвертая вообще не по силам базисному аппарату развития Ады (то есть для Ады это – нереальная абстракция).

Абстракция от имени. Достаточно ввести функцию с параметром и результатом нужного типа.

Что значит «нужного типа»? Пока мы абстрагируемся только от имени вектора, сохраняя все остальные его конкретные характеристики. Поэтому нужен тип, класс значений которого – 10-элементные векторы. Объявим его:

```
type вектор is array(1..10) of INTEGER;
```

Теперь нетрудно объявить нужную функцию.

```
function «-»(X : вектор) return вектор is
   Z: вектор;
begin
   for j in(1..10) loop
      Z(j) :=- X(j);
   end loop;
   return Z;
end «-»;
```

Обратите внимание, такая функция перекрывает предопределенную операцию «–». Становится вполне допустимым оператор присваивания вида

```
A := -A;
```

где A – объект типа «вектор», а знак «–» в данном контексте обозначает не предопределенную операцию над числами, а определенную нами операцию над векторами.

> **Замечание** (о запрете на новые знаки операций). В Аде новые знаки операций вводить нельзя. Это сделано для того, чтобы синтаксический анализ текста программы не зависел от ее смысла (в частности, от результатов контекстного анализа). Скажем, знак I нельзя применять для обозначения новой операции, а знак «–» можно.
>
> Именно для того, чтобы продемонстрировать перекрытие знака «–», мы ввели функцию, а не процедуру, хотя в данном случае последнее было бы эффективнее. Действительно, ведь наш исходный фрагмент программы создает массив-результат, изменяя массив-аргумент. А функция создает новый массив, сохраняя аргумент неизменным. Так что более точной и эффективной была бы абстракция-процедура следующего вида:

```
procedure минус(X : in out вектор) is
begin
   for j in(1..10) loop
      X(j) := – X(j);
   end loop;
end минус;
```

Уже при такой слабой абстракции (только от имени вектора) мы оказались перед необходимостью согласовывать операционную абстракцию и абстракцию данных (обратите внимание на диапазон 1...10 и в объявлении функции, и в объявлении типа). Так что принцип согласования абстракций работает и при создании языковых конструктов (при создании ЯП, на метауровне), и на уровне их применения. При этом согласование на метауровне призвано всячески облегчать согласование на уровне применения.

Вопрос. Нельзя ли упростить последнее в нашем случае?

Подсказка. Следует полнее использовать тип данных.

За счет согласованного объявления управляющей переменной цикла и диапазона допустимых индексов мы повысили надежность программы. Применяя функцию «–», невозможно выйти за границы массива – ведь ее аргументами могут быть только 10-элементные векторы.

Абстракция от длины вектора (начало). Пойдем дальше по пути абстракции. Как написать функцию, применимую к вектору любой длины? Уникальность типа требует снабдить определенным типом каждый параметр. Поэтому возникают два согласованных вопроса (снова действует принцип согласования абстракций!): «как объявить нужный тип?» и «как написать тело процедуры, работающей с массивом произвольной длины?».

Здесь полезно на время оторваться от нашего примера и вспомнить об общем контексте, в котором эти вопросы возникли.

4.6.4. Квазистатический контроль

Мы продолжаем заниматься в основном данными – одной из трех важнейших абстракций программирования. Исходя из потребности прогнозирования и контроля поведения объектов (в свою очередь выводимой из более общей потребности писать надежные и эффективные программы), мы пришли к концепции уникальности типа данных. А исходя из назначения системы типов, выделили динамические, статические и относительно статические (говоря короче, квазистатические) ЯП в зависимости от степени гибкости прогнозирования-контроля. Отметили, что в Аде концепция собственно типа ориентирована на прогнозирование-контроль статических характеристик поведения объектов, а концепция подтипа – на прогнозирование-контроль квазистатических (или, если угодно, квазидинамических) характеристик.

Наша ближайшая цель – обосновать полезность концепции подтипа.

Вспомним классификацию данных и в ней – фактор изменчивости. Он играет особую роль, так как касается самого динамичного атрибута объекта – его значения.

Поэтому с фактором изменчивости в наибольшей степени связано противоречие между потребностью гибко изменять поведение объектов и потребностью прогнозировать это поведение. Другими словами, это противоречие между потребностью в мощных операционных абстракциях, применимых к весьма разнообразным данным, и потребностью ограничить их применение, чтобы достичь надежности и эффективности.

Первая потребность требует свободы, вторая – дисциплины. Чем точнее удается прогнозировать изменчивость, чем в более жесткие рамки «зажимается» возможное поведение, тем надежнее контроль, больше возможностей экономить ресурсы. Но вместе с этим снижается общность, растут затраты на аппарат прогнозирования-контроля, растут затраты на создание близких по назначению программ, на их освоение.

Мы уже сталкивались с этим противоречием, когда занимались проблемой полиморфизма. Но тогда речь шла о небольшом наборе разновидностей (типов) объектов, и противоречие удалось разрешить за счет конечного набора операций с одним и тем же именем (каждая для своего набора типов объектов). Теперь нас интересует такая ситуация, когда разновидностей неограниченно много. Именно такая ситуация создалась при попытке абстрагироваться от длины вектора в программе «минус».

Абстракция от длины вектора (продолжение). В чем конкретно противоречие?

С одной стороны, нужна как можно более мощная операционная абстракция, применимая к векторам любой длины. Для этого необходимо иметь возможность ввести согласованный с такой абстракцией обобщенный тип, значениями которого могут быть массивы произвольной длины. Иначе возникнет противоречие с концепцией уникальности типа (**какое?**).

Такой тип в Аде объявить можно. Например, так:

```
type вектор_любой_длины is array(INTEGER range < >) of
INTEGER;
```

Вместо конкретного диапазона индексов применен оборот вида

```
тип range < >
```

который и указывает на то, что объявлен так называемый НЕОГРАНИЧЕННЫЙ регулярный тип, значениями которого могут быть массивы с любыми диапазонами индексов указанного типа (в нашем случае – целого).

Аналогичный неограниченный регулярный тип с диапазонами индексов перечисляемого типа вводит объявление

```
type таблица is array(буква range < >) of INTEGER;
```

Значения такого типа могут служить, например, для перекодировки букв в целые числа.

Упражнение. Напишите соответствующую программу перекодировки.

Вернемся к типу вектор_любой_длины. Как объявлять конкретные объекты такого типа? Ведь объявление вида

```
Y : вектор_любой_длины;
```

не прогнозирует очень важной характеристики объекта Y – его возможной длины (другими словами, прогнозу не хватает точности с точки зрения распределения ресурсов программы).

Поэтому само по себе такое объявление не позволяет ни обеспечить эффективность (нельзя распределить память при трансляции), ни настроить функцию на конкретный аргумент такого типа. И конечно, раз длина вектора не объявлена, то нет оснований контролировать ее (например, в процессе присваивания).

4.6.5. Подтипы

Чтобы разрешить указанное противоречие, авторы Ады были вынуждены ввести концепцию ПОДТИПА (специально для квазистатического прогнозирования-контроля изменчивости объектов). Основная идея – в том, чтобы при необходимости можно было, с одной стороны, удалить некоторые атрибуты объектов из сферы статического прогнозирования-контроля, не указывая их при объявлении типа, а с другой – оставить эти атрибуты для динамического прогнозирования-контроля с помощью подтипов.

Подтип представляет собой сочетание ТИПА и ОГРАНИЧЕНИЯ на допустимые значения этого типа. Значения, принадлежащие подтипу, должны, во-первых, принадлежать классу значений ограничиваемого типа и, во-вторых, удовлетворять соответствующему ОГРАНИЧЕНИЮ.

Подтип можно указывать при объявлении объектов. Например,

```
A : вектор_любой_длины(1..10);
```

объявляет 10-элементный вектор A (причем использовано так называемое ОГРАНИЧЕНИЕ ИНДЕКСОВ);

```
выходной : день_недели range сб..вс;
```

объявляет объект типа день_недели, принимающий значение либо «сб», либо «вс» (причем применяется так называемое ОГРАНИЧЕНИЕ ДИАПАЗОНА).

Бывают и другие виды ограничений (для вещественных и вариантных комбинированных типов).

Раньше мы говорили, что объявление объекта связывает с ним некоторый тип. На самом деле правильнее сказать, что оно связывает с ним некоторый подтип. Когда ограничение отсутствует, то все значения типа считаются удовлетворяющими подтипу.

Подтип, указанный в объявлении объекта, характеризует его во всей области действия объекта, в течение всего периода его существования. Поэтому становится возможным, во-первых, учитывать подтип при распределении памяти для объекта (например, для массива A выделить ровно десять квантов памяти); во-вторых, контролировать принадлежность подтипу при присваивании. Последнее приходится

иногда делать динамически (поэтому и идет речь о «квазистатическом контроле»). Это может замедлять работу программы, зато повышает надежность.

Пусть, например, объявлены объекты

```
A,B : вектор_любой_длины(1..10);
выходной : день_недели range сб..вс;
праздник : день_недели;
день_рождения : день_недели;
C,D : вектор_любой_длины(1..11);
будний_день : день_недели range пн..пт;
учебный_день : день_недели range пн..сб;
```

Тогда присваивания

```
A := B; B := A; праздник := день_рождения;
день_рождения := будний_день;
праздник := выходной;
C := D; D := C;
```

не потребуют никакой дополнительной динамической проверки, так как допустимые значения выражений в правой части присваивания всегда удовлетворяют подтипам объектов из левой части.

Присваивания

```
A := C; C := A; A := D; B := D; D := A; D := B;
будний_день := выходной;
выходной := будний_день;
```

также не требуют дополнительной динамической проверки – они всегда недопустимы, и обнаружить это можно статически, при трансляции.

А вот присваивания

```
будний_день := учебный_день; будний_день := праздник;
учебный_день := выходной; учебный_день := праздник;
```

нуждаются в динамической проверке (**почему?**).

4.6.6. Принцип целостности объектов

Вернемся к нашей процедуре «минус», вооруженные концепцией подтипа. Допустим, что ее параметр станет типа вектор_любой_длины. Как обеспечить настройку на конкретный вектор-аргумент? Другими словами, абстракцию мы обеспечили (есть обобщенный тип), а вот реальна ли она (чем обеспечена конкретизация)?

Вспомним, как это делается в Алголе-60 или Фортране. Границы конкретного массива-аргумента нужно передавать обрабатывающей процедуре в качестве дополнительных аргументов. Это и неудобно, и ненадежно (где гарантия, что будут переданы числа, совпадающие именно с границами нужного массива?).

Другими словами, перед нами – пример нарушения целостности объекта. Состоит оно в том, что цельный объект-массив при подготовке к передаче в качестве параметра приходится разбивать на части (имя – отдельно, границы – отдельно), а в теле процедуры эти части «собирать» (к тому же при полном отсутствии конт-

роля – ведь транслятор лишен информации о связи между границами и именем массива; знает о ней лишь программист).

Создатели более современных ЯП руководствуются принципом (сохранения) целостности объектов. Суть его в том, что ЯП должен обеспечивать возможность работать с любым объектом как с единым целым (не требуя дублировать характеристики объекта и тем самым устраняя источник ошибок). Более точно этот принцип можно сформулировать так:

вся необходимая информация об объекте должна быть доступна через его имя.

Соблюдение принципа целостности требует включения в базис соответствующих средств доступа к характеристикам объектов. В Аде, где этот принцип положен в основу языка, предопределены универсальные функции, позволяющие узнавать атрибуты конкретных объектов по именам этих объектов.

Они так и называются – АТРИБУТНЫЕ ФУНКЦИИ. Тот или иной набор атрибутных функций связывается с объектом в зависимости от его типа. В частности, для объектов регулярного типа определены атрибутные функции нигр(k) и вегр(k), сообщающие нижнюю и верхнюю границы диапазона индексов по k-му измерению. Например:

```
A'нигр(1) = 1,  B'нигр(1) = 1,
A'вегр(1) = 10, C'нигр(1)= 1,
D'вегр(1) = 11.
```

Абстракция от длины вектора (окончание). Теперь совершенно ясно, как объявить процедуру «минус», применимую к любому массиву типа вектор_любой_длины.

```
procedure минус(X : in out вектор_любой_длины) is
begin
   for j in(X'нигр(1)..X'вегр(1)) loop
      X(j) :=X(j);
   end loop;
end минус;
```

Для одномерных массивов вместо нигр(k) и вегр(k) можно писать короче – нигр и вегр, так что заголовок цикла может выглядеть красивее:

```
for j in(X’нигр..X’вегр)  loop .
```

Итак, мы полностью справились с нашей второй абстракцией. При этом воспользовались принципом целостности, чтобы обеспечить реальность абстракции. Со стороны данных для этого понадобилось ввести неограниченные типы и ограничения-подтипы, а со стороны операций – атрибутные функции (опять потребовалось согласовывать абстракции!).

> Сочетание относительно «свободных» типов с постепенным ограничением изменчивости вплоть до полной фиксации значений объектов (когда они становятся константами) широко применяется при прогнозировании-контроле поведения объектов в Аде. Подтип играет роль, аналогичную той роли, которую играл сам тип в ранних ЯП, – ведь подтип затрагивает только совокупность значений объектов, не касаясь применимых к ним операций (кроме присваивания, как было показано совсем недавно).

4.6.7. Объявление подтипа

Подтип, вводимый при объявлении объекта, является анонимным. Но можно объявлять именованные подтипы. Естественно делать это тогда, когда одни и те же ограничения нужно использовать для объявления многих различных объектов. Вот несколько примеров именованных подтипов:

```
subtype рабочий_день is день_недели range пн..пт;
subtype натуральный is INTEGER range 0.. INTEGER 'последний;
subtype положительный is INTEGER range 1..INTEGER'последний;
subtype цифра is CHARACTER range '0'..'9';
```

В качестве простого упражнения объявите подтип весенний_месяц, выходной_день, восьмеричная_цифра и т. п.

По внешнему виду объявление подтипа похоже на объявление производного типа. Однако это конструкты совершенно разного назначения. Разберемся с этим подробнее.

4.6.8. Подтипы и производные типы. Преобразования типа

Формально отличие производных типов от подтипов должно быть уже известно. Производный тип – это именно новый тип, характеризуемый классом значений и набором базовых операций.

По сравнению с родительским типом класс значений производного типа может быть сужен (за счет ограничения при объявлении производного типа), а набор базовых операций расширен (за счет объявлений базовых операций в определяющем производный тип пакете).

А для подтипа по сравнению с базовым типом набор базовых операций может быть только сужен (за счет ограничения допустимых значений).

Кроме того, производный тип в общем случае несовместим по присваиванию и передаче параметров как с родительским типом, так и с другими производными типами одного родительского типа. Например:

```
type год is new INTEGER range 0..2099;
type этаж is new INTEGER range 1..100;
A: год; B: этаж;
…
A := B; — недопустимо!
(Несовместимость типов, хотя значения заведомо попадут в нужный диапазон.)
```

Имя подтипа служит сокращением для сочетания ограничиваемого типа (назовем его БАЗОВЫМ ТИПОМ) и ограничения. Когда такое имя используется при объявлении объекта, считается, что объявлен соответственно ограниченный объект базового типа. Когда такое имя применяется в спецификации параметра процедуры или функции, то аргументом может быть любой объект базового типа, удовлетворяющий соответствующему ограничению. Объектам различных подти-

пов одного и того же базового типа можно присвоить любое значение базового типа, лишь бы оно было из подкласса, выделяемого подтипом левой части.

Содержательные роли, которые призваны играть в программе объекты различных подтипов одного базового типа (при квалифицированном использовании ЯП), должны быть аналогичными, взаимозаменяемыми (с точностью до ограничений на значения).

Другое дело – содержательные роли производных типов одного итого же родительского типа. Производные типы вводятся именно для того, чтобы можно было контролировать, применяются ли они точно по назначению. Поэтому объекты различных производных типов в общем случае не считаются взаимозаменяемыми и по присваиванию (а также сравнениям) несовместимы (компилятор обязан это контролировать). Содержательная роль различных типов, производных от одного и того же родительского типа, выражается в том, что им соответствуют совершенно различные наборы операций.

Вместе с тем при необходимости между такими (родственными) типами допустимы явные преобразования типа.

Лес типов. Назовем **лесом типов** ориентированный граф, вершинами которого служат типы, а дуги соединяют родительский тип с производным (рис. 4.1).

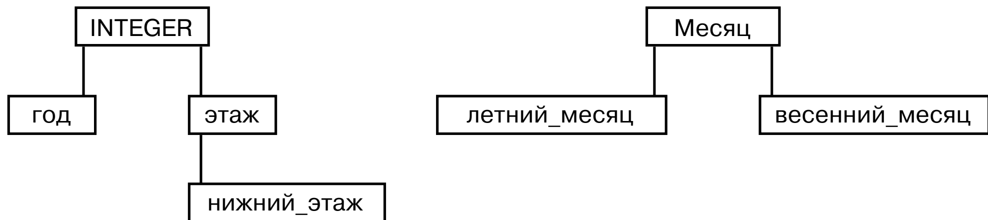


Рис. 4.1
где type нижний_этаж is new этаж range 1..3;
type летний_месяц is new месяц range июнь..август;
type весенний_месяц is new месяц range март..май

Как видите, лес типов в программе может состоять из отдельных деревьев.

Родственные типы и преобразования между ними. Типы из одного дерева в лесу типов называются РОДСТВЕННЫМИ. В Аде допустимы явные преобразования между родственными типами, которые указываются с помощью имени так называемого целевого типа, то есть типа, к которому следует преобразовать данное. Каждое определение производного типа автоматически (по умолчанию) вводит и операции преобразования родственных типов (но применять эти операции нужно явно!). Например, можно написать

```
A := год(B);
```

а также

```
B := этаж(A);
```

Обе эти операции (и «год», и «этаж») введены по умолчанию указанными выше объявлениями производных типов, но применены явно, когда потребовалось присвоить объект одного родственного типа другому. Во втором присваивании потребуется проверить попадание в диапазон допустимых значений.

Преобразования родственных типов (с учетом различных содержательных ролей объектов разных типов) далеко не всегда оправданы. Именно поэтому и вводятся явные преобразования, которые программист должен указать сознательно. Так надежность согласуется с гибкостью. По существу, в таких случаях программист явно говорит, что **полученное значение должно в дальнейшем играть другую содержательную роль**.

Например, в задаче моделирования учреждения может появиться тип «сотрудник», характеризуемый целой анкетой атрибутов и набором базовых операций («загружен_ли», «выполнить_задание», «включить_в_группу» и т. п.). Есть и

```
type рук_группы is new сотрудник;
```

со своими базовыми операциями («дать_задание», «подготовить_план_работы», «где_сотрудник» и т. п.).

Пусть объявлены объекты

```
A: сотрудник;
B: рук_группы;
```

Тогда присваивание

```
B := рук_группы(A);
```

содержательно может означать «повышение» сотрудника А. Ясно, что «автоматически» такое преобразование не делается!

4.6.9. Ссылочные типы (динамические объекты)

До сих пор в наших примерах встречались лишь статические и квазистатические объекты. Время существования статических объектов совпадает с полным временем выполнения всей программы. Время существования квазистатических объектов согласуется со временем очередного исполнения их статически определимой области действия.

Область действия объекта – это часть текста программы, при выполнении которой объект считается доступным (например, для чтения или записи его значения, для передачи объекта в качестве параметра и т. п.).

Например, в Паскале область действия (локальной) переменной – процедура или функция, где эта переменная объявлена. В Аде (квази)статические объекты создаются при исполнении объявлений и исчезают в момент завершения исполнения соответствующей области действия.

Так как квазистатические объекты создаются и исчезают синхронно с исполнением фиксированных компонент текста программы (областей действия этих объектов), то в каждой из них легко явно назвать все квазистатические объекты

индивидуальными именами, фигурирующими непосредственно в тексте программы (то есть также квазистатическими). Это еще один признак квазистатического объекта – такие и только такие объекты могут иметь квазистатические имена.

Но имена содержательных обрабатываемых объектов далеко не всегда удобно фиксировать в тексте программы. Ведь эти объекты могут находиться во внешней для программы среде или возникать динамически как во внешней среде, так и в самой программе при обработке других объектов. Такие динамически возникающие объекты называют динамическими объектами. Иногда динамическими объектами называют и (квази)статические объекты с динамически изменяющимися размерами или структурой.

Обычный прием, обеспечивающий доступ к динамическим объектам, состоит в том, что ссылки на такие объекты служат значениями квазистатических объектов, явно фигурирующих в программе. Когда ЯП не предоставляет адекватных выразительных средств, динамические объекты приходится моделировать с помощью квазистатических объектов или их частей. Например, на Фортране динамические очереди, списки, таблицы, стеки приходится моделировать (квази)статическими массивами.

Таким образом, возникает технологическая потребность в категории так называемых ССЫЛОЧНЫХ ТИПОВ, то есть типов данных, класс значений которых – ссылки на динамические объекты.

Динамические объекты отличаются от статических или квазистатических, во-первых, тем, что создаются при выполнении так называемых ГЕНЕРАТОРОВ, а не при обработке объявлений; во-вторых, тем, что доступ к ним осуществляется через объекты ссылочных типов.

Поэтому и время существования динамических объектов в общем случае связано не с местом их первого упоминания в программе, а со временем существования ссылки. Ссылку можно передавать от одного ссылочного объекта другому, сохраняя динамический объект даже при исчезновении ранее ссылавшихся на него квазистатических объектов.

Чтобы сохранить возможность квазистатического прогнозирования-контроля поведения динамических объектов, необходимо классифицировать ссылочные объекты в соответствии с типом динамических объектов, на которые им разрешено ссылаться. Однако концепция типа не обязана различать динамические и квазистатические объекты, так как с точки зрения класса значений и применимых операций динамический объект не отличается от квазистатического (кроме операции создания и, возможно, уничтожения).

Так и сделано в Аде (как и в Паскале), где объекты одного и того же типа могут быть как статическими, так и динамическими в зависимости от того, объявлены они или созданы генератором. Вместе с тем с каждой ссылкой жестко связан тип, на объекты которого ей разрешено ссылаться.

4.7. Типы как объекты высшего порядка. Атрибутные функции

Когда тип данных легко обозначать и определять, естественно появляется потребность обращаться с ним самим как с данным, употреблять его в качестве операнда различных операций без каких-либо априорных ограничений.

4.7.1. Статическая определимость типа

Однако одно ограничение касается всех статических ЯП (в том числе и ЯП Ада) – его можно назвать **статической определимостью типа**: тип каждого объекта должен быть определим по тексту программы. Ведь только при этом условии тип может играть роль основного средства статического прогнозирования-контроля.

Из статической определимости следует, что тип объектов, связанных с какой-либо операцией, определяемой пользователем, не может быть ее динамическим параметром (а значит, и аргументом), а также не может быть результатом нетривиальной операции.

Вопрос. Почему?

Подсказка. Все определяемые пользователем операции работают в период исполнения программы.

Формально сказанное не исключает операций над типами как значениями, однако содержательно этими операциями нельзя воспользоваться динамически при соблюдении принципа статической определимости типа.

4.7.2. Почему высшего порядка?

Статическая определимость типа – необходимое условие статического контроля типов. Но контролировать типы – значит применять к ним «контролирующие» операции (при статическом контроле – в период компиляции!). Так что с точки зрения потребности в контроле тип следует считать объектом высшего порядка (метаобъектом), который служит для характеристики «обычных» объектов (низшего порядка).

4.7.3. Действия с типами

Что же можно «делать» с таким объектом высшего порядка, как тип данных?

Во-первых, можно (и нужно) получать из одних типов другие, то есть преобразовывать типы (не объекты данных, а именно типы). Преобразователями типов служат конструкторы типов. Так, конструктор производного типа преобразует родительский тип в новый тип, в общем случае ограничивая класс значений и сохраняя

набор базовых операций. Конструктор регулярного и комбинированного типов преобразует типы компонент регулярных и комбинированных объектов в новый составной тип. Определяющий пакет представляет собой преобразователь объявленных в нем типов, дополняющий набор их базовых операций. Обратите внимание, все названные преобразователи типов (иногда говорят «типовые» функции с ударением на первом слоге) в язык Ада встроены. Они не только предопределены создателями ЯП, но и по виду своего вызова отличаются от обычных функций. По смыслу это, конечно, функции, аргументами и результатами которых служат типы. Но синтаксически они функциями не считаются и работают не в период исполнения, как остальные функции, а в период трансляции (то есть это «статические» функции).

Таким образом, в преобразователях типов используются все три компоненты типа. Имя – для идентификации; множества значений и базовых операций – как аргументы для получения новых множеств значений и операций. Вместе с тем статическая (то есть ограниченная) семантика преобразователей типа подчеркивает ограниченность концепции типа в языке Ада (подчеркивает ее «статичность» и согласуется с ней). Такая ограниченность характерна для всех статических ЯП, где тип играет роль основного средства статического прогнозирования-контроля.

Во-вторых, с помощью типов можно управлять. Тип управляет прогнозированием-контролем, когда используется в объявлении объектов и спецификации параметров. Тип непосредственно управляет связыванием спецификации и тела процедуры, когда работает принцип перекрытия. Тип управляет выполнением цикла, когда непосредственно определяет диапазон изменения управляющей переменной цикла. В перечисленных примерах тип в Аде по-прежнему выступает в качестве аргумента предопределенных конструктов. Пользователь лишен возможности ввести, например, новый принцип прогнозирования-контроля, новый принцип перекрытия или новую разновидность циклов.

В-третьих, тип может служить аргументом функций, вычисляющих отдельные значения того же самого или другого типа. Это и есть АТРИБУТНЫЕ функции. Например, функция «первый» вычисляет наименьшее значение заданного дискретного типа, функция «длина» вычисляет целочисленное значение – длину диапазона заданного дискретного типа. На самом деле аргументом таких функций служит ПОДТИП (ведь тип – частный случай подтипа). А подтип, как мы видели в примере с процедурой «минус», может быть связан с объявленным объектом. Поэтому в качестве аргумента атрибутной функции может выступать не только тип, но и объект данных, который в таком случае и идентифицирует соответствующий подтип.

Так что уже использованные нами функции «нигр» и «вегр» считаются атрибутными – их аргументами может быть любой регулярный подтип. Обозначения всех атрибутных функций предопределены в Аде. Чтобы отличать их от обычных функций (точнее, чтобы объекты высшего порядка – подтипы – не оказывались в синтаксической позиции динамических аргументов), и применяется специфическая запись вызова атрибутных функций – аргумент-подтип отделяют апострофом.

Еще одна, четвертая возможность использовать тип как аргумент – настройка РОДОВЫХ СЕГМЕНТОВ.

Вопрос. Можно ли атрибутную функцию запрограммировать на Аде?

Ответ. Если не использовать других атрибутных функций, то в общем случае нельзя – нет примитивных операций над типами. Некоторые атрибутные функции можно выразить через другие с помощью родовых сегментов.

4.8. Родовые (настраиваемые) сегменты

Мы уже отмечали, что тип в Аде не может быть динамическим параметром. Однако статическая определимость типов не противоречит статической же обработке заготовок пакетов и процедур, с тем чтобы настраивать их (в период трансляции) на конкретные значения так называемых РОДОВЫХ («СТАТИЧЕСКИХ») ПАРАМЕТРОВ.

Статическими параметрами заготовок пакетов и процедур (РОДОВЫХ СЕГМЕНТОВ) могут поэтому быть и типы, и процедуры. Определяя родовой сегмент, можно ввести абстракцию, пригодную для использования в различных конкретных контекстах.

В Аде имеется мощный аппарат статической абстракции-конкретизации – аппарат родовых сегментов. Приведем пример родового пакета с четырьмя родовыми параметрами. После ключевого слова generic следуют четыре спецификации родовых параметров:

```
generic
   type элемент is private;
   -- допустим, любой тип (кроме ограниченного приватного)
   type  индекс is(< >);
   -- допустим, любой дискретный тип
   type  вектор is array(индекс) of элемент;
   -- любой регулярный тип, но имя типа индексов указывать обязательно нужно!
   with function сумма(X, У: элемент) return элемент;
   -- закончился список из четырех формальных родовых параметров,
      последний параметр – формальная функция «сумма», применимая к объектам
      формального типа   «элемент»
package на_векторах is – «обычная» спецификация пакета
   function сумма(A, B: вектор) return вектор;
   function сигма(A: вектор) return элемент;
end на векторах;
```

Обратите внимание, здесь одна функция «сумма» – формальный родовой параметр, другая – обычная функция, объявленная в спецификации пакета. Как писать тело такого пакета, должно быть понятно:

```
package body   на_векторах is
   function сумма(A,B: вектор) return вектор is
```

```
      Z: вектор;
   begin
      for j in вектор'нигр .. вектор'вегр loop
         Z(j) :=сумма (A(j), B(j));
      end loop;
      return Z;
   end сумма;
   function сигма(A: вектор) return элемент is
      Z: элемент := A (вектор'нигр);
   begin
      for j in вектор'нигр + 1 .. вектор'вегр loop
         Z :=сумма(Z, A(j));
      end loop;
      return Z;
   end сигма;
end на_векторах;
```

Вот возможная конкретизация этого пакета:

```
package на_целых_векторах is new на_векторах (INTEGER, день, ведомость, '+');
```

Здесь тип «ведомость» считается введенным объявлением

```
type ведомость is array(день range < >) of INTEGER;
```

а '+' – предопределенная операция для целых.

Так что если

```
T: ведомость(Вт..Пт) := (25,35,10,20);
R: ведомость(Вт..Пт) := (10,25,35,15);
```

то в соответствующем контексте

```
сумма(T,R) = (35,60,45,35); сигма(T) = 90; сигма(R) = 85;
```

Обратите внимание на применение агрегатов, а также на то, как в них используется линейный порядок на дискретных типах.

Родовые аргументы должны строго соответствовать спецификации родовых параметров. За этим ведется строгий контроль. Так, функция '+' подошла, а, например, «or» или тем более «not» не подойдет **(почему?)**.

> **Замечание.** Обратите внимание на нарушение принципа целостности объектов в аппарате родовых сегментов.
>
> **Вопрос.** В чем это проявляется?
>
> **Подсказка.** Разве все аргументы конкретизации не связаны еще в объявлении типа «ведомость»? Зачем же заставлять программиста дублировать эту связь?

Указывать категорию и некоторые другие характеристики родовых параметров нужно для контроля за использованием параметра внутри родового сегмента. В нашем случае, например, внутри этого сегмента недопустимы какие-либо операции с объектами типа «элемент», кроме явно определяемых программистом (например, «сумма»), а также присваиваний и сравнений. С другой стороны, в качестве типов-аргументов в данном случае допустимы любые типы (кроме

ограниченных приватных). В общем случае в качестве спецификации родового параметра можно указывать категории перечисляемых типов, целых, вещественных, ссылочных, регулярных, но не комбинированных.

Вопрос. Почему недопустимы комбинированные?

Подсказка. Иначе пришлось бы в родовом сегменте фиксировать названия и типы полей. Кстати, чем это плохо?

4.9. Числовые типы (модель числовых расчетов)

4.9.1. Суть проблемы

Рассматривая основные технологические потребности, невозможно обойти потребность вести числовые расчеты. Эта потребность, как известно, в свое время предопределила само возникновение ЭВМ. Хотя сейчас потребность в числовых расчетах – далеко не самая главная в развитии компьютеров и ЯП, абсолютная потребность в объеме, точности и надежности числовых расчетов продолжает расти. Так что ни в одном базовом языке индустриального программирования ее игнорировать нельзя.

Парадоксально, что так называемые машинно независимые языки для научных расчетов (Фортран, Алгол и их диалекты) не предоставили удовлетворительной модели числовых расчетов, в достаточной степени независимой от программной среды.

В этом их аспекте особенно сказалась несбалансированность средств абстракции и конкретизации. Абстрагироваться от особенностей среды можно, а настроиться на конкретную среду – нельзя. Обеспечить надежность при изменении среды – проблема.

Суть в том, что ни в Фортране, ни в Алголе нет возможности явно управлять диапазоном и точностью представления числовых данных. Можно лишь указать, что требуется «двойная точность» (в некоторых диалектах градаций больше), но какова эта точность, зависит от реализации. Таким образом, пользователь «машинно независимого» ЯП оказывается в полной зависимости от конкретной среды, если ему нужно гарантировать надежность расчетов.

Одна из причин такой ситуации – в том, что в начале «эры ЭВМ» скорости числовых расчетов придавалось исключительно большое значение. Поэтому считалось практически нереальным применять какое-либо представление чисел и какие-либо базовые операции, отличные от непосредственно встроенных в машину. Поскольку такие встроенные числовые типы различны на различных ЭВМ, считалось невозможным эффективно решить проблему выбора представления в соответствии с машинно независимыми указаниями пользователя. Поэтому проблема обеспечения надежности числовых расчетов традиционно оставалась вне рамок «машинно независимых» языков.

По мере накопления пакетов программ и осознания того факта, что зависимость от конкретных представлений числовых типов – одно из важнейших препятствий при переносе программ из одной вычислительной среды в другую, рос интерес к созданию достаточно универсальной схемы управления числовыми расчетами.

К моменту создания Ады проблема надежности программного обеспечения (в частности, при переносе из одной вычислительной среды в другую) была осознана как важнейшая, потеснившая по своей значимости проблему скорости расчетов. К тому же доля числовых расчетов в общем времени исполнения программ существенно сократилась. Появились и методы относительно эффективной реализации вычислений с любой заданной точностью (за счет микропрограммирования, например).

Все это сделало актуальной и реальной попытку разработать гибкую модель управления числовыми расчетами. Одна из таких моделей воплощена в Аде.

Управлять представлением числовых данных можно и в языке ПЛ/1, и в Коболе. Однако в этих ЯП отсутствует явная связь представления данных с гарантией надежности расчетов. В частности, отсутствуют машинно независимые требования к точности реализации предопределенных операций над числами. Эти требования – основная «изюминка» модели управления расчетами в Аде.

4.9.2. Назначение модели расчетов

Необходимо, чтобы при работе с числовыми типами программист мог гарантировать надежность расчетов независимо от целевой среды (если только эта среда пригодна для их выполнения). Гарантировать надежность означает, в частности, гарантировать как необходимую точность расчетов, так и отсутствие незапланированных исключительных ситуаций (переполнение, исчезновение порядка) при допустимых исходных данных. Очевидно, что достичь такой цели можно, только предоставив программисту возможность объявлять нужные ему диапазон и точность представления чисел.

Искусство авторов языка (создателей модели управления расчетами) проявляется в том, чтобы найти разумный компромисс между единообразием управления и эффективностью расчетов при гарантированной надежности.

4.9.3. Классификация числовых данных

Начать естественно с разумной классификации числовых данных в зависимости от характера расчетов. В Аде три категории числовых типов: целые, вещественные плавающие и вещественные фиксированные. Для каждой категории типов имеются свои средства объявления.

Целые типы предназначены для точных расчетов, вещественные – для приближенных. При этом плавающие типы воплощают идею действий с нормализованными числами, представленными с относительной погрешностью, зависящей от количества значащих цифр, а фиксированные – идею действий с ненормализо-

ванными числами, представленными с абсолютной погрешностью, зависящей от положения десятичной точки.

Плавающие типы чаще встречаются в ЯП для числовых расчетов. Поэтому ограничимся демонстрацией основной идеи управления расчетами на примере плавающих типов Ады.

4.9.4. Зачем объявлять диапазон и точность

Оставим пока в стороне синтаксис и семантику соответствующих объявлений. Ведь объявить диапазон и точность – дело относительно нехитрое. А вот что делать, когда разрядная сетка или принятое в компьютере представление чисел не соответствует объявлению типа? Важно понимать, что объявление типа остается особенно полезным именно в такой ситуации.

Действительно, в худшем случае исполнитель получает всю информацию, чтобы признать (и информировать пользователя), что он непригоден для такой программы. Если объявленные требования к диапазону и точности критичны для предлагаемых расчетов, то такой отказ, безусловно, предпочтительнее, чем трата времени и усилий на заведомо ошибочные расчеты, да еще с риском оставить пользователя в «счастливом» неведении. К тому же легко автоматически или визуально выделить фрагменты программы, вызвавшие непригодность исполнителя. Это помогает либо подобрать подходящий исполнитель (автоматически, если доступна, например, неоднородная сеть машин), либо изменить выделенные компоненты программы.

4.9.5. Единая модель числовых расчетов

Чтобы гарантировать надежность расчетов в любой среде, то есть управлять диапазоном и точностью на достаточно высоком уровне абстракции, пользователь должен опираться на модель расчетов, единую для всех сред. Однако, стремясь в первую очередь к переносимости программ, нельзя забывать про эффективность конкретизации (вспомните принцип реальности абстракций). Другими словами, единая модель расчетов должна быть гибкой (требовать от конкретных реализаций минимума свойств и действий).

В Аде абстракция (единство) обеспечивается понятием модельных чисел, а конкретизация (гибкость) – понятием допустимых чисел.

Совокупность модельных чисел каждого подтипа полностью фиксируется на уровне, независимом от среды (то есть определяется семантикой ЯП и применяемым программистом объявлением).

Допустимые числа – это некоторое надмножество модельных чисел, зависящее от конкретной реализации (рассчитанной на конкретную целевую среду).

Гибкость модели проявляется в том, что расчеты программируются и оцениваются с помощью модельных чисел, а выполняются с допустимыми числами.

Из набора предопределенных числовых типов реализация имеет право подобрать для заданного объявления числового типа такой (по возможности мини-

мальный) базовый тип допустимых чисел, который удовлетворяет указанным при объявлении ограничениям диапазона и точности.

Итак, с одной стороны, все реализации ориентированы на единую «систему координат» – модельные числа; с другой – каждая реализация работает со своими допустимыми числами.

Рассмотрим на примере управление диапазоном модельных чисел. Пусть объявлен плавающий тип

```
type скорость is digits 8; – число значащих цифр(D) = 8
```

После ключевого слова digits указан спецификатор точности D (равный в нашем случае восьми), определяющий диапазон модельных чисел типа «скорость» по следующим единым для всех реализаций правилам.

Прежде всего необходимо вычислить количество B двоичных цифр, обеспечивающих ту же относительную погрешность, что и D десятичных цифр. В общем случае

```
B = целая_часть(D * ln(10)/ln(2) + 1)
```

то есть приблизительно 3,3 двоичной цифры для представления одной десятичной. В нашем случае

```
B = [ 8 * 3,3 + 1 ] = 27.
```

Диапазон модельных чисел определяется как совокупность всех (двоичных!) чисел, представимых в виде

```
знак * мантисса * (2 ** порядок),
```

где знак – это +1 или –1, мантисса – правильная двоичная дробь, записанная ровно B двоичными цифрами, первая из которых 1 (то есть нормализованная дробь), порядок – целое число между –4*B и +4*B.

Таким образом, с каждой спецификацией D связывается конечный набор вещественных модельных чисел. При этом фиксируется не только количество цифр в мантиссе, но и максимальный порядок. В нашем случае двоичный порядок равен

```
27 * 4 = 108.
```

Соответствующий десятичный порядок 4*D = 32. Чем больше порядок, тем «реже» встречаются модельные числа – точность представления плавающих типов относительна.

4.9.6. Допустимые числа

Как уже сказано, диапазон допустимых чисел – это расширение диапазона модельных чисел, самое экономичное с точки зрения реализации. В принципе, в нашем случае в качестве допустимых чисел могут фигурировать, например, числа с 40-разрядной мантиссой и 7-разрядным порядком. Так что для представления такого диапазона допустимых чисел подойдет, например, 48-разрядное машинное слово. На практике транслятор подберет в качестве базового для типа «скорость» ближайший из предопределенных плавающих типов REAL, SHORT_REAL, LONG_REAL или иной из плавающих типов, определяемый реализацией.

Уточнить диапазон и точность при объявлении производного типа, числового подтипа или объекта можно, как обычно, с помощью ограничения. Например:

```
type высота is new скорость range 0.0 .. 1.0E5;
(высота может меняться от нуля до десяти тысяч).
subtype высота_здания is высота range 0.0 .. 1.0E3;
высота_полета : высота digits 5;
(для переменной высота_полета допустимая точность меньше, чем в типе «высота»).
```

4.10. Управление операциями

Управление диапазоном и точностью для плавающих типов имеется во многих ЯП. Однако этого мало для надежного и независимого от среды управления расчетами, если программист лишен возможности на уровне модельных чисел учитывать погрешности, вносимые предопределенными (то есть элементарными арифметическими) операциями. Такой возможности до Ады не предоставлял ни один ЯП. Результаты всех предопределенных операций над допустимыми числами определяются в Аде с помощью модельных чисел следующим единым для любой среды способом.

Модельным интервалом называется любой интервал между модельными числами. С каждым допустимым числом ассоциируется так называемый связанный модельный интервал – это минимальный модельный интервал, к которому принадлежит рассматриваемое число (для модельных чисел связанным интервалом считается само модельное число!).

Основное требование к точности реализации предопределенных операций (модельное ограничение) состоит в том, что результат операций над допустимыми числами должен попадать в минимальный модельный интервал, содержащий точные математические результаты рассматриваемой операции при изменении аргументов операции в пределах их связанных интервалов.

Наглядно это можно представить следующим рисунком (рис. 4.2).

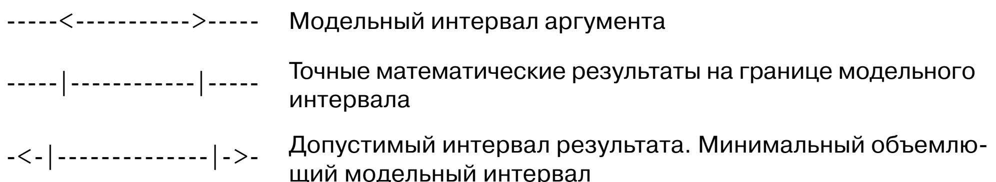


Рис. 4.2

Таким образом, значениями операций вещественных типов в конкретных реализациях могут быть любые числа, обладающие модельным интервалом, но отклонения результатов операций регламентированы модельным ограничением.

Искусство программиста состоит теперь в том, чтобы гарантировать надежность расчетов, подбирая диапазон и точность в соответствии с условиями задачи.

При этом он не должен забывать, что при реализации излишний диапазон или излишняя точность могут стоить дорого и по времени, и по памяти (а могут и превысить возможности реализации). Короче говоря, **нужно программировать как можно ближе к нуждам решаемой задачи**.

В Аде предусмотрено, что реализация может предоставлять несколько предопределенных (именованных или анонимных) плавающих типов с различными диапазонами и точностью расчетов.

Искусство автора компилятора проявляется в том, чтобы компилятор был в состоянии подобрать подходящий предопределенный тип (в качестве допустимых чисел) для любого (или почти любого) объявления типа, с оптимальными затратами на расчеты при полной гарантии соответствия модельному ограничению.

Обратите внимание, в Аде отдано явное предпочтение двоичным машинам (ведь допустимые числа включают модельные, причем в соответствии с модельным ограничением результат операции над модельным числом должен быть точным, если математический результат оказывается модельным числом; для элементарных (!) арифметических операций такое требование выполнимо, в отличие от произвольных математических функций). К тому же не любым двоичным машинам, а с достаточно большим порядком во встроенном представлении плавающих чисел (ведь далеко не во всех машинах допустимы порядки, вчетверо превышающие длину мантиссы; во всяком случае в БЭСМ-6 это не так).

4.11. Управление представлением

Суть проблемы. Единая модель числовых расчетов, как мы видели, позволяет программисту непосредственно управлять представлением данных числовых типов в целевой машине. Но потребность управлять представлением возникает не только для числовых типов и не только для данных. Причем требуется управлять в некотором смысле окончательным представлением (как и в случае числовых типов), а не промежуточным (когда, например, данные приватного типа мы представляли данными комбинированного типа). Другими словами, требуется управлять представлением объектов в терминах понятий, в общем случае выходящих за рамки машинно независимого ЯП и определенных только на конкретной целевой машине.

Такая потребность особенно часто возникает в системных программах, вынужденных взаимодействовать, в частности, с нестандартными устройствами обмена (ввода-вывода) или со средствами управления прерываниями целевой машины. Например, требуется явно указать адрес, с которого располагается реализация реакции на конкретное прерывание.

Конечно, программы, использующие управление окончательным (или, как часто говорят, «абсолютным») представлением, перестают быть машинно независимыми. Однако это та мера зависимости от машины (та мера конкретизации), без которой программа неработоспособна.

Вместе с тем это зависимость только от целевой машины. Сами средства управления представлением данных могут оставаться нормальными конструктами машинно независимого ЯП, где они играют роль компонент общего аппарата связывания, а именно связывания абстрактной спецификации данных с их конкретным абсолютным представлением. Такое связывание выполняется при трансляции и может быть выполнено на любой транслирующей (инструментальной) машине.

С другой стороны, если связывание такого рода выделено как достаточно общее языковое понятие и ему соответствует легко идентифицируемый конструкт, то настройка программы (ее перенос) на другую целевую машину сводится к изменению только таких конструктов.

Именно такой идеей и руководствовались авторы ЯП Ада. Управление абсолютным представлением выделено в конструкт, который называется УКАЗАНИЕМ ПРЕДСТАВЛЕНИЯ (спецификацией представления – representation clauses). Указание представления должно следовать за объявлением тех сущностей, представление которых в нем конкретизируется. В Аде можно указывать представление для типов, объектов данных, подпрограмм, пакетов и задач, а также для входов.

Подход к языковым конструктам как компонентам единого аппарата абстракции-конкретизации позволяет легко обнаружить перспективу развития средств управления представлением в Аде. А именно высшим уровнем оформления абстракции представления было бы выделение специального программного сегмента («модуля представления»), предназначенного исключительно для описания привязки машинно независимых сегментов к конкретной целевой машине.

Вопросы. Что бы это дало? Каковы недостатки такого решения?

Интересно, что эта перспектива неоднократно рассматривалась автором на лекциях за несколько лет до того, как ему стало известно о воплощении модулей управления представлением в системе Кронус (новосибирской модификации Модулы-2). Так что анализ ЯП с самых общих «философских» позиций действительно позволяет иногда прогнозировать их развитие.

Рассмотрим несколько примеров одного из самых нужных указаний представления – УКАЗАНИЯ АДРЕСА.

Указание адреса применяют для того, чтобы связать объект данных, подпрограмму или вход задачи с той ячейкой памяти целевой машины, которая играет особую роль. Это может быть регистр определенного периферийного устройства; адрес, по которому передается управление при определенном прерывании; ячейка, состояние которой определяет режим работы машины, и т. п.

Чтобы можно было записать указание адреса, должен быть доступен тип «адрес». Содержательно значения этого типа играют роль адресов памяти целевой машины, формально это один из предопределенных типов. В Аде его определяющим пакетом считается предопределенный пакет «система», характеризующий целевую машину. Следовательно, тип «адрес» становится доступным с помощью указателя контекста вида

```
with система; use система;
```

Вот примеры указания адреса с очевидным назначением:

```
for управл_ячейка use at 16#0020#;
```

после at записана шестнадцатеричная константа типа «адрес». Такое указание адреса должно быть помещено среди объявлений блока, пакета или задачи после объявления объекта управл_ячейка.

```
task обработка_прерывания is
   entry выполнить;
   for выполнить use at 16#40#;
      -- вызвать вход «выполнить» — это значит передать управление в ячейку 16#40#
end обработка_прерывания;
```

Еще примеры использования указаний представления:

```
слово : constant := 4;
-- элемент памяти — байт, "слово" — из четырех байтов.
type состояние is(A,M,W,P);
-- четыре характеристики состояния:
      -- каков код символов(ASCII или EBCDIC);
      -- разрешены ли прерывания;
      -- ждет ли процессор;
      -- супервизор или задача
type маска_байта is array(0..7) of BOOLEAN;
type маска_состояния is array (состояние) of BOOLEAN;
type маска_режима is array(1..4) of BOOLEAN;
type слово_состояние_программы is record
   маска_системы : маска_байта;
   ключ_защиты : INTEGER range 0..3;
   состояние_машины : маска_состояния;
   причина_прерывания : код_прерывания;
   код_длины_команды : INTEGER range 0..3;
   признак_результата : INTEGER range 0..3;
   маска_программы : маска_режима;
   адрес_команды : адрес;
end record;

   -- ниже следует указание представления для этого типа
for слово_состояния_программы use record at mod 8;
   -- адреса записей указанного типа
   -- должны быть нулями по модулю 8, то есть
   -- адресами двойных слов. Далее указаны требования
   -- к расположению полей записи относительно ее начала
   маска_системы at 0 * слово range 0..7;
   -- маска системы расположена в первом байте двойного слова.
   ключ_защиты at 0 * слово range 10..11;
   -- разряды 8 и 9 не используются.
   состояние_машины at 0 * слово range 12.. 15;
   причина   прерывания at 0 * слово range 16..31;
   код_длины_команды at 1 * слово range 0..1;
   признак_результата at 1 * слово range 2 ..3;
```

```
    маска_программы at 1 * слово range 4 .. 7;
    адрес_команды at 1 * слово range 8..31;
end record;
```

Здесь применено так называемое УКАЗАНИЕ ПРЕДСТАВЛЕНИЯ ЗАПИСИ. Запись типа слово_состояние_программы располагается в двойном слове, то есть по адресам, кратным 8, причем для каждого поля указано расположение относительно начала записи с точностью до разряда.

Итак, машинно независимые языковые конструкты, примером которых служат средства управления представлением в Аде, позволяют в машинно независимой форме указывать машинно зависимые характеристики программ. Это один из аспектов ЯП, которые позволяют отнести Аду к «машинно ориентируемым» (но машинно независимым!) ЯП.

Вопрос. Какие еще аспекты Ады можно отметить в этой связи?

Подсказка. В Аде немало свойств, «определяемых реализацией», например некоторые свойства атрибутных функций.

4.12. Классификация данных и система типов Ады

Рассматривая данные как одну из основных абстракций программирования, мы выделили семь факторов их классификации. Опираясь на эту классификацию, дадим краткий обзор средств управления данными в Аде с учетом выделенных технологических потребностей.

1. Содержательные роли данных. На первый взгляд, возможность явно отражать в программе содержательные роли данных так, чтобы можно было автоматически контролировать связывания, кажется почти фантастической. Теперь мы знаем, что идея очень проста. Во-первых, нужно, чтобы содержательная роль объекта получила имя, отличающее ее от других ролей. Во-вторых, проектируя объект данных, нужно связывать с ним имя той роли, в которой он должен выступать при выполнении программы. Имя роли естественно указывать при объявлении объекта. В-третьих, проектируя содержательные действия, нужно явно указывать при их объявлении имена ролей объектов, участвующих в этих действиях.

При таком прогнозировании контроль за соответствием поведения объектов объявленным их ролям становится легко формализуемым. Его обеспечивает концепция типа, ориентированная на имена. В частности, реализованная в Аде концепция уникальности типа.

2. Строение данных. Классификация по структуре данных имеется практически во всех ЯП. В Аде по этому фактору различаются скалярные, регулярные, комбинированные, ссылочные, задачные и приватные данные. Соответственно, выделяются и категории типов данных. Правила объявления типов данных в Аде таковы, что к одному типу можно относить только данные, формально «близкие»

по своему строению. При необходимости подчеркнуть, что данные разного строения играют одинаковую содержательную роль, их можно объявить в качестве вариантов так называемого вариантного комбинированного типа.

3. Изменчивость данных. При объявлении типа данных в Аде всегда сообщается максимально допустимая изменчивость данных этого типа. Но когда объявляют отдельный объект или целый класс объектов (подтип), изменчивость можно ограничить. Концепция типа в Аде в целом обеспечивает квазистатическое управление изменчивостью.

4. Способ определения. Различаются предопределенные типы и объявляемые пользователем. С этой точки зрения в Аде особенно интересны приватные типы. На уровне использования они инкапсулированы и могут быть сделаны неотличимыми от предопределенных. Без приватных типов можно вводить новые операционные абстракции, но не абстракции данных.

5. Представление. Ада позволяет управлять, во-первых, относительным представлением данных, когда речь идет о представлении приватных типов на уровне их реализации типами иных категорий; во-вторых, абсолютным представлением, когда речь идет о представлении любых типов на целевой машине (посредством указаний представления).

6. Внешние свойства. Набором применимых операций в Аде управляют посредством объявления типа и определяющего пакета.

7. Управление доступом. Доступом в Аде управляют с помощью приватных типов, приватной части, а также разделения спецификации и реализации пакета. Используют также указатель контекста with, указатель сокращений use, блочную структуру и другие средства.

Вопрос. Какие, например?

Подсказка. Ссылочные типы. А еще?

Итак, система типов языка Ада хорошо согласуется с рассмотренной классификацией. С другой стороны, эта классификация указывает направления развития адовских средств управления данными.

Упражнение. Предложите такие средства.

Подсказка. Уже упоминавшиеся модули представления, а также более тонкое управление доступом, полномочиями, наследованием и т. п.

Наша классификация отражает характеристики данных, обычно охватываемые концепцией типа. Но данные различаются и по другим факторам. Один из них – отношение данного и модуля программы. Очень четко такое отношение отражено в языке Том [7] понятием класса данного. Выделены глобальные данные, параметры, локальные и синхропараметры. Аналогичные понятия имеются, конечно, и в других ЯП.

Вопрос. Как вы думаете, разумно ли объединить указанное понятие класса и типа?

Подсказка. Не забудьте, в частности, о концепции уникальности типа.

На этом закончим знакомство с системой типов в Аде. Читателя, заинтересованного в углубленном изучении проблем, связанных с типами, отсылаем к увлекательной книге А. В. Замулина [9].

4.13. Предварительный итог по модели А

Итак, выделив три основные абстракции – данные, операции и связывание, мы углубились в основном в изучение первой из них, одновременно получая представление о мощном языке индустриального программирования (фактически мы строим «максимальную» модель такого языка – модель А).

В отличие от традиционных ЯП (Фортрана, Алгола, Бейсика и др.), язык Ада ориентирован скорее на данные, чем на операции. В нем в первую очередь поддерживается такой стиль программирования, когда проектируется не столько программа, сколько комплекс программных услуг, опирающийся на ключевую структуру данных.

Решая наши модельные задачи в таком стиле, мы одновременно вникали в суть основных абстракций программирования. Параллельно мы знакомились с различными видами операций (для каждой категории типов – свои) и с различными видами связывания (статическое, динамическое, квазистатическое, подразумеваемое по умолчанию, выбираемое компилятором, указываемое программистом).

Вне поля нашего зрения осталось еще несколько важных абстракций, которыми мы теперь и займемся, одновременно завершая как знакомство с языком Ада, так и построение нашей модели А.

Упражнение. Приведите примеры перечисленных видов связываний.

Вопрос. Как вы думаете, чем отличается модель А от языка Ада?

Глава 5

Раздельная компиляция

5.1. Понятие модуля 146
5.2. Виды трансляций 146
5.3. Раздельная трансляция 146
5.4. Связывание трансляционных модулей.. 147
5.5. Принцип защиты авторского права 148

5.1. Понятие модуля

Одна из важнейших концепций ЯП – концепция модульности. Ей посвящена обширная литература (полезно почитать, в частности, [12]). В самом широком смысле модуль – это абстракция от контекста, доведенная до воплощения в отдельном объекте, пригодном для хранения, обработки и использования в разнообразных контекстах с минимальными накладными расходами. Образно говоря, модуль проще заимствовать, чем создавать заново (иногда последнее может быть даже запрещено авторским правом).

В связи с тем, что трансляция играет особую роль при использовании ЯП, особую важность приобретают различные концепции трансляционных модулей, то есть оформленных по специальным правилам фрагментов текста на ЯП, пригодных для относительно независимой от контекста трансляции и последующего использования без полной перетрансляции. Относительность указанной независимости от контекста проявляется в том, что в общем случае для трансляции модуля могут в той или иной степени требоваться фрагменты потенциального контекста. Обычно эти фрагменты находятся в **трансляционной библиотеке** в форме **трансляционных модулей**, содержащих определения имен, используемых, но не определенных в рассматриваемом модуле.

5.2. Виды трансляций

Различают так называемую «**цельную**» трансляцию (модульность отсутствует – транслятору предъявляют всю программу целиком; примером служат стандартный Паскаль и первые версии Турбо Паскаля), **раздельную трансляцию** (предъявляют модуль и трансляционную библиотеку – примером служат Ада и Модула-2), «**шаговую**», или «**инкрементную**», трансляцию (транслятору предъявляют лишь очередное дополнение или исправление к программе – примером служит инструментальная система для создания Ада-программ на специализированном компьютере R1000) и, наконец, «**независимую**» трансляцию (транслятору предъявляют только один модуль, а связывание оттранслированных модулей выполняется редактором связей или загрузчиком – примером служит Фортран).

У каждого вида трансляции свои преимущества и недостатки, довольно очевидные.

Вопрос. Какие?

5.3. Раздельная трансляция

Раздельная трансляция (компиляция) модулей – одна из критичных технологических потребностей индустриального программирования. Без нее практически невозможно создавать сколько-нибудь значительные по объему программы (**почему?**).

В Аде ТРАНСЛЯЦИОННЫЙ МОДУЛЬ (или просто МОДУЛЬ) – это программный сегмент, пригодный для раздельной трансляции. Иначе говоря, это фрагмент текста, который можно физически отделить от контекста и применять посредством ТРАНСЛЯЦИОННОЙ БИБЛИОТЕКИ.

5.4. Связывание трансляционных модулей

Трансляционные модули связываются для того, чтобы взаимодействовать как части единой программы. При этом приходится называть имена партнеров по связыванию.

Выделим два основных вида связывания, которые назовем односторонним и двусторонним соответственно. При одностороннем связывании лишь один из двух связываемых модулей называет имя своего партнера, при двустороннем – оба. В стандартном Паскале и Бейсике трансляционных модулей нет, однако процедуры вполне можно считать модулями (но не трансляционными, а логическими). В Фортране имеются настоящие трансляционные модули (которые так и называются – «модули», а иногда «программные единицы»). Имеются трансляционные модули и во всех современных диалектах Паскаля (unit в Турбо Паскале начиная с версии 4.0, а также в проекте нового стандарта ИСО). В перечисленных случаях применяется только одностороннее связывание модуля с контекстом (в нужном контексте указывается имя требуемого модуля, но в самом модуле явные ссылки на контекст отсутствуют; например в процедуре не перечисляются ее вызовы).

5.4.1. Модули в Аде

Трансляционный модуль в Аде – это такой программный сегмент, все внешние связи которого оформлены как связи с трансляционной библиотекой. Для трансляционных модулей применяются оба вида связывания. С односторонним связыванием мы уже фактически познакомились, когда применяли указание контекста (with). Например, трансляционными модулями были: спецификация пакета управление_сетями, процедура построение_сети, родовой сегмент «очередь». Все это примеры так называемых первичных, или открытых (library), модулей. Название «открытых» отражает факт, что в силу односторонней связи этими модулями можно пользоваться открыто, в любом месте программы, для чего достаточно употребить соответствующее указание контекста.

Тело пакета управление_сетью и вообще тела подпрограмм и пакетов без внешних связей (кроме связи со «своей» спецификацией) служат примерами так называемых вторичных модулей. Повторение в них тех же имен, что и в первичных модулях, можно трактовать как указание двусторонней связи с соответствующими спецификациями. Ведь на тела можно ссылаться извне только через спецификации. При этом имя пакета или подпрограммы в заголовке спецификации тракту-

ется как указание на связь с соответствующим телом, а то же имя в заголовке тела трактуется как указание на связь с соответствующей спецификацией. Так что связь действительно двусторонняя.

Когда же сами спецификации представляют собой внутренние компоненты других модулей, то по-прежнему можно оформлять соответствующие таким спецификациям тела пакетов, процедур и задач как вторичные модули, но для этого нужно явно указать уже двустороннюю связь. Именно: в том модуле, где находится спецификация, применяют так называемую заглушку, указывающую на вторичный модуль, а в заголовке вторичного модуля явно указывают имя того модуля, где стоит заглушка. Признаком двусторонней связи служит ключевое слово separate.

Например, можно оформить как вторичный модуль тело любой процедуры из пакета управление_сетями. Выберем процедуру «вставить» и функцию «перечень_связей». Тогда в теле пакета вместо объявления этих подпрограмм нужно написать заголовки вида

```
function перечень_связей(узел: имя_узла) return BOOLEAN is
separate;
procedure вставить(узел: in имя_узла) is separate;
```

Перед нами две ссылки на вторичные модули, две заглушки. Соответствующие вторичные модули следует оформить так:

```
separate(управление_сетью); — указано, где находится заглушка для этой функции
function перечень_связей(узел: имя_узла) return BOOLEAN is
   ... — тело как обычно
end перечень_связей;

separate(управление_сетью) — указано, где находится заглушка для этой процедуры
procedure вставить(узел: in имя_узла) is
   ... — тело как обычно
end вставить;
```

Теперь вторичные модули «перечень_связей» и «вставить» можно транслировать отдельно. В Аде предписан определенный (частичный) порядок раздельной трансляции. В соответствии с ним все вторичные (secondary) модули следует транслировать после модулей с соответствующими заглушками (то есть после «старших» модулей, которые, в свою очередь, могут быть как первичными, так и вторичными).

Итак, ко вторичному модулю можно «добраться» только через его партнера. Поэтому, в отличие от открытых библиотечных модулей, их естественно называть закрытыми. Свойство закрытости обеспечено применением явной двусторонней связи.

5.5. Принцип защиты авторского права

Подчеркнем, что закрытые модули появились в Аде не случайно. Они, с одной стороны, естественное развитие разделения абстракции и конкретизации (спецификации и реализации) до уровня модульности (ведь модуль – это материальное

воплощение абстракции). С другой – они обслуживают важный принцип конструирования языка Ада, который можно было бы назвать **принципом защиты авторского права** (языковыми средствами).

Дело в том, что тексты вторичных модулей можно вовсе не предоставлять пользователю при продаже программных изделий, созданных на Аде. Ему передаются только спецификации – открытые модули, а также защищенные от чтения оттранслированные вторичные. Пользователь никак не сможет незаконно добраться до вторичных модулей. При соответствующей защите от несанкционированного доступа он не только не сможет неправильно пользоваться ими, но не сможет и скопировать или употребить для построения других систем (отдельно от закупленных).

Принцип защиты авторского права получил существенное развитие в идеологии так называемого объектно-ориентированного программирования, восходящей еще к ЯП Симула-67 и в своем современном виде воплощенной в таких новейших ЯП, как Смолток, Оберон, Си++ и Турбо Паскаль 5.5 (подробности – в разделе «Объектно-ориентированное программирование»).

Замечание. С учетом идеологии конкретизирующего программирования возникает соблазн рассматривать раздельно транслируемый модуль как программу, управляющую транслятором (интерпретируемую транслятором) в процессе раздельной трансляции. Результатом выполнения такой программы может быть модуль загрузки, перечень обнаруженных ошибок и (или) изменения в трансляционной библиотеке. При таком общем взгляде на раздельную трансляцию управление ею становится столь же разнообразным и сложным, как программирование в целом. Такой взгляд может оказаться полезным, когда универсальный модуль рассматривается как источник (или генератор) разнообразных версий программы, учитывающих особенности конкретных применений.

Разнообразными становятся и способы извлечения из внешней среды информации об указанных особенностях (параметры, анализ среды, диалог с программистом и т. п.). Так что в общем случае создание универсальных модулей в корне отличается от написания частей конкретных программ. Такие модули становятся метапрограммами, описывающими процесс или результат создания частей конкретных программ. Поэтому язык для написания универсальных модулей и управления их связыванием в общем случае может сильно отличаться от исходного языка программирования (хотя, с другой стороны, есть немало аргументов в пользу их совпадения). Рассмотренный общий взгляд по существу выводит за рамки собственно раздельной трансляции.

Настройка «универсальных» модулей на конкретное применение в Аде есть – ее обслуживает аппарат родовых сегментов. Но конкретизация родовых сегментов выполняется не при связывании собственно модулей, а позже, в рамках последующей трансляции сегментов.

Глава 6

Асинхронные процессы

6.1. Основные проблемы 152
6.2. Семафоры Дейкстры 155
6.3. Сигналы 157
6.4. Концепция внешней дисциплины 159
6.5. Концепция внутренней дисциплины: мониторы 160
6.6. Рандеву 164
6.7. Проблемы рандеву.................. 165
6.8. Асимметричное рандеву 166
6.9. Управление асимметричным рандеву (семантика вспомогательных конструктов) 167
6.10. Реализация семафоров, сигналов и мониторов посредством асимметричного рандеву 169
6.11. Управление асинхронными процессами в Аде 172

6.1. Основные проблемы

До сих пор мы занимались последовательным программированием и соответствующими аспектами ЯП. Этот вид программирования характеризуется представлением о единственном исполнителе, поведение которого и нужно планировать. Во всяком случае, это поведение всегда можно было мыслить как некоторый **процесс** – последовательность действий из репертуара этого исполнителя, строго упорядоченных во времени.

Основной стимул для расширения сферы наших интересов состоит в том, что реальный мир весьма разнообразен и сложен, так что его моделирование одним последовательным процессом часто оказывается неадекватным.

Поэтому займемся языковыми средствами, предназначенными для планирования поведения некоторого **коллектива исполнителей**. Мы употребляем слово «коллектив», а не просто «совокупность» исполнителей, чтобы подчеркнуть, что нас интересуют, как правило, только взаимодействующие исполнители, совместно решающие некоторую проблему. Поведение каждого из них будем по-прежнему мыслить как некоторый процесс, но будем считать, что поведение коллектива в целом в общем случае никакой строго упорядоченной во времени последовательностью действий не описывается.

Однако оно естественным образом описывается совокупностью (коллективом) взаимодействующих последовательных процессов. Так что соответствующие языковые средства должны предусматривать описание такого коллектива процессов (в частности, их запуск и завершение) и (самое сложное, важное и интересное) описание их взаимодействия. Подчеркнем, что если взаимодействие процессов отсутствует, то коллектив распадается на отдельные процессы, задача управления которыми решается в последовательном программировании.

Подчеркнем, что коллективы исполнителей сплошь и рядом встречаются в реальной жизни: сам компьютер естественно представлять коллективом исполнителей-компонент (центральный процессор, память, периферия), сети компьютеров – также коллектив исполнителей; производственные, военные, информационные процессы естественно группировать в коллективы. Наконец, в некоторых задачах коллективы процессов возникают в целях оптимизации (матричные операции, сеточные задачи и т. п.).

Как уже сказано, упорядоченность отдельных событий (фактов поведения) в разных процессах часто можно считать несущественной для задачи, решаемой всем коллективом. Поэтому обычно такой упорядоченностью можно пренебречь и считать процессы асинхронными, в частности исполняемыми параллельно на подходящем коллективе процессоров (реальных или виртуальных).

По этой причине соответствующее программирование называют «параллельным», а соответствующие ЯП – параллельными, или **языками параллельного программирования** (ЯПП).

С другой стороны, когда упорядоченностью событий в отдельных процессах пренебречь нельзя, программа должна ее явно или неявно предписывать, для чего

в ЯПП должны быть соответствующие **средства синхронизации асинхронных процессов**. Важнейшая цель синхронизации – организация обмена информацией между исполнителями (асинхронными процессами), без которого их взаимодействие невозможно.

Итак, нас будут интересовать особенности ЯПП и прежде всего средства синхронизации и обмена как частные случаи средств взаимодействия процессов.

Основные проблемы (кроме обычных для последовательного программирования) фактически названы. Это **синхронизация**, **обмен** (данными), **запуск процессов**, **завершение процессов**.

Наиболее интересные работы в области ЯПП прямо или косвенно имеют следующую перспективную цель: выработать систему понятий, позволяющую концентрироваться на решении специфических проблем параллельного программирования тогда и там, когда и где это существенно для решаемой задачи, а не для выбранных средств программирования. Другими словами, цель состоит в том, чтобы **программировать асинхронные процессы со степенью комфорта, не уступающей лучшим достижениям последовательного программирования**.

Эта цель еще впереди, но есть ряд достижений, с которыми мы и познакомимся.

Наша ближайшая цель – продемонстрировать как пользу, так и проблемы параллельного программирования, а главное – серию языковых средств, применяемых для решения этих проблем: семафоры, сигналы, мониторы, рандеву, каналы, а также связанные с ними языковые конструкты.

В качестве основного примера выберем характерную для параллельного программирования задачу о поставщике и потребителе некоторой информации.

Задача о поставщике и потребителе

Итак, нужно описать два процесса-соисполнителя, один из которых (поставщик) вырабатывает некоторое сообщение и поставляет его потребителю, который, в свою очередь, получает сообщение и потребляет его в соответствии с назначением (которое нас не интересует).

Предполагается, что эти два исполнителя способны работать совершенно независимо, кроме ситуаций, когда они непосредственно заняты обменом информацией между собой.

Вопрос о том, что значит «вырабатывает» и «потребляет», нас также в этой задаче не интересует, поскольку касается в каждом случае только одного из процессов и, следовательно, укладывается в знакомые рамки последовательного программирования. Зато вопрос о том, как понимать слова «поставляет» сообщение и «получает» сообщение, касается самой сути взаимодействия наших соисполнителей.

Различным вариантам уточнения этих двух слов и будет посвящена серия наших попыток решить задачу. После некоторых колебаний выбран скорее исторический, чем логический порядок изучения возможных подходов к ее решению. Хотя, усвоив предыдущий материал, читатель подготовлен к восприятию современных решений, полезно проникнуться проблемами первопроходцев, тем более что рассматриваемые при этом средства так называемого «нулевого уровня» (семафоры и сигналы) используются и в современных языках (например, в Смолтоке).

Необходимость уточнения смысла слов «поставляет» и «получает» вызвана тем, что они подразумевают некоторый способ взаимодействия (в остальном совершенно независимо работающих) процессов.

Во всяком случае, это не может быть вызов обычных процедур «поставлять» и «получать» в соответствующем процессе хотя бы потому, что выполнение таких процедур (по самому их смыслу) предполагает определенное состояние готовности внешней для процесса среды. Если в этой среде нет ничего, кроме процесса-партнера (о состоянии которого в общем случае ничего не известно – ведь он работает асинхронно), то взаимодействие просто невозможно.

Итак, следует сконструировать внешнюю среду, допускающую взаимодействие асинхронных процессов.

Первое, что приходит в голову человеку, привыкшему к последовательному программированию, – считать процессы-исполнители работающими в общем внешнем контексте (среде), где им доступны общие переменные. Рассмотрим соответствующие решения.

Итак, первый вариант взаимодействия – через общие переменные. Точнее говоря, будем считать, что поставщик и потребитель программ обмениваются сообщениями через некоторый общий (доступный обоим партнерам) буфер, способный хранить ограниченное число сообщений. Подробности устройства буфера нас пока не интересуют. Однако ясно, что, если не принять дополнительных мер, состояние буфера будет столь же непредсказуемо, как и состояние процесса-партнера. Поэтому вся проблема – в том, как избежать этой непредсказуемости.

Представим некоторую начальную детализацию наших процессов и общего контекста. Будем писать в уже опробованном «адовском» стиле.

```
package общий is
. . .
b : буфер;
   procedure поставить(X : in сообщение);
   procedure получить(X : out сообщение);
. . .
end общий;

with общий; use общий;              with общий; use общий;
task поставщик  is;                 task потребитель is;
. . .                               . . .
   loop;                               loop;
      . . .                               . . .
      выработать(X);                      получить(X);
      поставить(X);                       потребить(X);
      . . .                               . . .
   end loop;                           end loop;
end поставщик;                      end потребитель;
```

Если считать, что процедуры «поставить» и «получить» соответственно заполняют и освобождают буфер b, то очевидно, что наша программа правильно работать не будет! Как уже сказано, состояние буфера непредсказуемо. В частности,

нельзя заносить сообщения в полный буфер и нельзя выбирать сообщения из пустого буфера. Поэтому нужны две функции «полон» и «пуст», сообщающие о состоянии буфера.

Но теперь будет непредсказуемой связь между значением такой функции и реальным состоянием буфера – ведь оно может изменяться сколь угодно быстро из-за асинхронных действий партнеров. Итак, нужна определенная синхронизация действий партнеров.

Именно состояние буфера не должно изменяться другим партнером, пока первый партнер узнает состояние буфера, принимает решение о посылке или выборке сообщения и выполняет это решение.

Важно понимать, что такая синхронизация может быть выполнена только с помощью средств, поставляемых внешним контекстом (внешней средой). Например, люди применяют для синхронизации с партнерами часы, видят партнеров, чувствуют и т. п.

Эти внешние средства должны удовлетворять определенным требованиям, чтобы обеспечивать корректность синхронизации. Во всяком случае, один партнер не должен иметь возможность мешать другому партнеру воспользоваться средством синхронизации. Это важнейшее требование корректности можно обеспечивать по-разному.

Основной принцип – **монополизация доступа** к соответствующему средству. Другими словами, если процесс обладает правом пользоваться средством синхронизации, то в период, когда он им реально пользуется, внешняя среда гарантирует его монополию на это средство, точнее, на пользование этим средством в определенной роли (в этот же период тем же средством в другой роли может пользоваться партнер, как в случае рандеву и каналов, – см. ниже).

Принцип монопольного доступа можно интерпретировать и так, что весь акт доступа считается неделимым, выполняемым мгновенно с точки зрения внутреннего времени процессов, что не позволяет другому процессу вмешиваться в исполнение этого акта.

Одно из назначений синхронизации – обеспечить монопольный доступ к общим переменным. Как видим, это возможно лишь за счет монопольного доступа к базовым средствам синхронизации.

6.2. Семафоры Дейкстры

Рассмотрим два вида средств синхронизации – (двоичные) семафоры и (двоичные) сигналы.

Их основное назначение отражено в названии: семафоры применяют для синхронизации прохождений процессами своих так называемых «критических участков» – вполне аналогично тому, как синхронизируют движение поездов, поднимая и опуская семафоры и обозначая тем самым занятость железнодорожного перегона.

Поезд допускается на перегон, если путь свободен (семафор поднят (открыт)), поезд ждет своей очереди, если путь занят (семафор опущен (закрыт)), и, нако-

нец, семафор закрывается, как только поезд выходит на перегон (занимает путь); семафор открывается, когда поезд покидает перегон (освобождает путь).

Внешняя среда, обеспечивающая управление процессами и семафорами, должна обеспечивать, в частности, приостановку и активизацию процессов, организацию их очереди к семафору и, конечно, монопольный доступ к семафорам (неделимость действий с семафорами).

Все эти возможности Дейкстра предложил концентрировать в двух операциях:

```
оградить (S) и освободить (S)
```

для объекта S типа «семафор», принимающего два значения («свободен» и «занят»).

Семантика этих операций такова:

Оградить(S): `if S=свободен then S:=занят else [приостановить текущий процесс и поставить его в очередь(S)].`

Освободить(S): `if пуста(очередь(S)) then S:=свободен else [возобновить процесс, первый в очереди(S)].`

Семантика семафоров приспособлена к такой дисциплине программирования, когда «критический участок» любого процесса предваряется операцией «оградить» и завершается операцией «освободить». Процесс, желающий монопольно распоряжаться некоторым ресурсом, охраняемым семафором S, первой операцией «закрывает за собой дверь» и не дает другим процессам себе мешать. Завершив свои дела, он второй операцией «открывает дверь» для других желающих.

Тем самым по отношению к общим (разделяемым) ресурсам реализуется **режим взаимного исключения с развязкой** – на своих критических участках процессы **взаимно исключают** доступ партнеров к ресурсу, а на остальных участках **совершенно развязаны** – никак не ограничивают действий партнеров. Понятно, что разделяемых ресурсов может быть много. Тогда для каждого из них следует заводить свой семафор.

В нашем случае такой ресурс один – буфер. Поэтому достаточно одного семафора (критические участки выделены):

```
package общий is
. . .
   b : буфер; — буфер сообщений.
   S : семафор; — с ним связаны соответствующие операции
   procedure поставить...;
   procedure получить...;
end общий;
```

```
with общий; use общий;
task поставщик  is;
. . .
   loop;
      выработать(X);
      оградить(S);
      поставить(X);
      освободить(S);
   end loop;
end поставщик;
```

```
with общий; use общий;
task потребитель is;
. . .
   loop;
      оградить(S);
      получить(X);
      освободить(S);
      потребить(X);
   end loop;
end потребитель;
```

Теперь партнеры не мешают друг другу в период доступа к буферу и вежливо уступают доступ, когда он им не нужен.

Однако программа все равно не будет работать корректно! **(Почему?)**

Дело в том, что партнеры не следят за состоянием буфера. Не следят сами и не помогают следить напарнику.

Можно было бы воспользоваться функциями «полон» и «пуст», сообщающими о состоянии буфера. Например, так (пишем только внутренние циклы):

```
loop;
   выработать(X);
   оградить(S);
   while полон loop
      ждать; — фикс. время
   end loop;
   поставить(X);
   освободить(S);
end loop;
```

```
loop;
   оградить(S);
   while пуст loop
      ждать;
   end loop;
   получить(X);
   освободить(S);
   потребить(X);
end loop;
```

Однако и такое решение неприемлемо и работать не будет. **(Почему?)**

Вложенные циклы с ожиданием могут долго работать... в монопольном режиме! Например, если буфер полон, то бессмысленно ждать внутри критического участка, пока он освободится, – ведь партнеру буфер недоступен, так как семафор закрыт! Это пример **тупика**.

Следует писать так:

```
while полон loop
   освободить(S);
   ждать;
   оградить(S);
end loop;
```

```
while пуст loop
   освободить(S);
   ждать;
   оградить(S);
end loop;
```

Теперь программа будет работать. **(Хорошо ли?)**

Не слишком хорошо. Внутренние циклы нерационально расходуют активное время процессора – это особенно неприятно при реализации всей системы на единственном физическом процессоре. Циклов **активного ожидания** в таком случае стараются избегать.

6.3. Сигналы

Избежать циклов активного ожидания можно, заменив его **пассивным ожиданием** (без занятия процессора), организуемым с помощью средств синхронизации, называемых сигналами.

Сигнал Е – это объект типа «сигнал», принимающий значения «есть» и «нет». Аналогично семафору с ним связаны очередь процессов, «ждущих сигнала Е», и две операции: послать (Е) и ждать (Е), – управляющие этой очередью. Семантика этих операций такова:

```
послать(E):

if пуста(очередь (E)) then E:=есть;
```

```
else [возобновить первый ("ждущий") процесс в очереди(E)];
```

ждать (E):

```
if E=есть then E:=нет;
else [приостановить текущий процесс и поместить его (последним) в очередь (E)].
```

Как видим, семантика сигналов двойственна семантике семафоров (послать = освободить, а ждать = оградить).

Однако если семафорами пользуются для того, чтобы в рамках одного процесса ограждать критические участки от возможного влияния других процессов, то сигналы используются именно для организации взаимного влияния процессов. С помощью сигналов партнеры могут сообщать информацию о событиях, становящихся им известными. С другой стороны, они могут пассивно (в очереди) ждать наступления этих событий.

В нашем примере таких событий два: неполнота и непустота буфера. Поэтому нужны два сигнала – непуст и неполон. Заметив, что цикл ожидания становится ненужным (ожидание обеспечивают средства синхронизации – за это и боролись!), напишем (теперь уже окончательную) схему нашей программы полностью:

```
package общий is
   . . .
      b: буфер; — для сообщений
      s: семафор;
   неполон, непуст: сигнал; — сигналы-будильники
   procedure поставить(X: in сообщение);
   procedure получить(X: out сообщение);
   function полон ...;
   function пуст...;
end общий;
```

```
task поставщик is
X: сообщение;
. . .
   loop;
      выработать(X);
      оградить(S);
         if полон then
            освободить(S);
            ждать(неполон);
            оградить(S);
         end if;
      поставить(X);
      освободить(S);
      послать(непуст);
   end loop;
   . . .
end поставщик;
```

```
task потребитель is
X: сообщение;
. . .
   loop;
      оградить(S);
      if пуст then
         освободить(S);
         ждать(непуст);
         оградить(S);
      end if;
      получить(X);
      освободить(S);
      потребить(X);
      послать(неполон);
   end loop;
   . . .
end потребитель;
```

Полезно подчеркнуть следующие существенные моменты.

1. Условный оператор развязывает действия партнеров. Без него была бы фактически полная синхронизация (то есть процессы не были ли бы фактически асинхронными).
2. Ограждение проверки нужно для монополизации разделяемого ресурса, а освобождение – чтобы избежать тупика. Ведь ожидаемый сигнал может прийти только от партнера – надо дать последнему возможность работать с буфером. При этом первый же цикл партнера обязательно даст такой сигнал, так что «застрять» на ожидании нельзя.
3. Семафор обеспечивает поочередную работу процессов – он не может быстро мигать в результате работы только одного процесса, когда второй ждет у семафора. При первом же освобождении пойдет второй процесс, и первый будет ждать у своего «оградить», если первым до него доберется.
4. Целостность объектов в нашей программе не обеспечена – связь семафора с буфером, а также сигнала с буфером никак в программе не отражена – отражена лишь в мыслях программиста. Это неадекватно (не отражает сути дела) и опасно, так как нет контроля за этими связями.
   Иными словами, свойства семафоров и сигналов как языковых конструктов не соответствуют основному критерию качества ЯП (усложняют программирование и понимание программ) .
5. Структурно не отделены части программы, существенно зависящие от взаимодействия с другими процессами, от частей, в которых можно абстрагироваться от такого взаимодействия (и самого факта управления одним из членов коллектива асинхронных процессов). Средства программирования таковы, что при написании буквально любой команды следует опасаться «подводных камней» параллелизма.

6.4. Концепция внешней дисциплины

Частично резюмируя пп. 4 и 5 из параграфа 6.3, частично обобщая их, можно сделать вывод об использовании в нашей программе (сознательно или интуитивно) так называемой «концепции внешней дисциплины» взаимодействия процессов. Термин «внешней» отражает отношение дисциплины к разделяемым процессами ресурсам.

Суть этой концепции – в том, что о дисциплине (правилах) использования разделяемых ресурсов должны заботиться сами процессы-партнеры. Другими словами, она «локализована» вне общих ресурсов. Примером такой дисциплины может служить «скобочное» правило применения операций «оградить» и «освободить».

Итак, мы рассмотрели пример задачи поставщик–потребитель и показали, как в рамках концепции внешней дисциплины при обмене справиться с проблемой порчи данных (немонопольный, множественный доступ), а при синхронизации – с проблемой порчи управления (тупики и лишние ожидания). При этом мы совершенно игнорируем запуск и завершение процессов.

Полезно понимать, что взаимно дополнительные свойства семафоров и сигналов позволяют сводить использование семафоров к использованию сигналов, и наоборот. Однако если при этом применение сигналов вместо семафоров допускает толкование, вполне согласующееся с названием соответствующих операций, то обратное неверно. Именно парно-скобочное применение операции оградить (S) ... освободить (S) естественно трактовать как

ждать (разрешения-ввести-в-критический-участок-для-S) и

послать (сигнал-о-завершении-критического-участка-для-S),

где в скобках указаны два сигнала, соответственно посылаемые и ожидаемые внешней (управляющей процессами) средой (точнее, некоторым процессом-диспетчером), в которой находится пара операторов

послать (сигнал-о-завершении-критического-участка-для-S) и

ждать (разрешения-ввести-в-критический-участок-для-S).

Так что один семафор сводится к двум сигналам. Свести к одному опасно! Иначе диспетчер будет равноправен с другими процессами. Здесь же только он имеет право послать (разрешение...).

Невозможность обратной замены (сигналов на семафоры) с сохранением «скобочного» смысла операции очевидна – ведь в процессе может оказаться только одна из таких операций. Однако если не сохранять симметрию операций в одном процессе, то и здесь заменяющее истолкование возможно: «ждать» трактуется как «оградить» последующие операторы, пока не будет сигнала (но не от вмешательства, а от исполнения), а «послать» – как «освободить» от вынужденного ожидания.

6.5. Концепция внутренней дисциплины: мониторы

Замысел внутренней дисциплины вполне укладывается в идеологию РОРИУС: разделяемый ресурс следует представить некоторым специальным комплексом услуг, реализация которого концентрирует в себе все особенности параллелизма и конкретной операционной среды, а использование становится формально совершенно независимым от поведения и даже наличия процессов-партнеров.

Здесь слово «формально» подчеркивает факт, что в процессе-пользователе оказывается невозможным обнаружить какие-либо следы присутствия процессов-партнеров. Более того, его семантика полностью описывается в терминах взаимодействия с одним только указанным специальным комплексом услуг. Хотя, конечно, эти услуги содержательно связаны с наличием и функционированием процессов-партнеров.

Итак, концепция внутренней дисциплины состоит в локализации всех средств управления взаимодействием коллектива процессов в рамках некоторого явно выделенного разделяемого ресурса – специального комплекса услуг, называемого **монитором**.

Мониторы могут быть весьма разнообразными. Содержательно близкие роли играют диспетчеры (супервизоры) операционных систем, но их далеко не всегда сознательно проектируют в рамках концепции внутренней дисциплины. Мы рассмотрим некоторую полезную абстракцию, так называемые мониторы Хансена-Хоара, предложенные в 1973–1975 гг.

Продолжим рассматривать нашу задачу-пример о поставщике и потребителе.

Ключевая идея: **разделяемый ресурс (буфер) следует превратить в комплекс услуг, самостоятельно обеспечивающий корректность доступа к буферу.** С подобной идеей мы уже знакомились в последовательном программировании – пакеты предоставляют комплекс услуг с защитой внутренних ресурсов от нежелательного доступа.

Теперь требуется аналог пакета в ЯПП. Пакет был в состоянии защитить ресурс за счет того, что доступ допускался только через разрешенные операции. Причем поскольку исполняемый процесс был единственным, **всегда исполнялась единственная операция пакета**. Если это фундаментальное свойство пакета сохранить, то для корректности доступа к ресурсу совершенно несущественно, сколько и каких операций выполняется вне пакета.

Таким образом, пакет с указанным встроенным фундаментальным свойством операций (встроенным по семантике соответствующего ЯПП) пригоден и для обслуживания асинхронных процессов-пользователей. Такой пакет и называется монитором Хансена-Хоара. (Он был воплощен, в частности, в ЯП Параллельный Паскаль Бринча Хансена.)

Другими словами, монитор – это пакет со встроенным режимом взаимного исключения предоставляемых пользователю операций. Тем самым реализуется монопольный доступ процесса к ресурсу без каких-либо специальных указаний со стороны этого процесса (и усилий программиста).

Полезно осознать, что здесь мы в очередной раз имеем дело с рациональной абстракцией и удобной конкретизацией. Монитор позволяет пользователю отвлечься от «параллельной природы» использования ресурса и работать с ним в обычном последовательном (монопольном) режиме. Вместе с тем он позволяет программисту при создании тела монитора полностью учесть «природу» конкретного ресурса и доступа к нему.

Сказанное следует понимать именно так, что пользователь может программировать, полностью игнорируя параллелизм процессов. Этот идеал пользования разделяемым ресурсом-буфером в случае нашего примера выглядит так:

```
with буф; use буф;                    with буф;    use буф;
task поставщик is                     task потребитель is
   . . .                                 . . .
      loop                                  loop
         выработать(X);                        получить(X);
         поставить(X);                         потребить(X);
      end loop;                             end loop;
end поставщик;                        end потребитель;
```

Здесь «буф» обозначает нужный контекст. Им и служит монитор, предоставляющий необходимые услуги. Достигается полная абстракция от способа обмена.

Обратите внимание, наш идеал в точности совпадает с первоначальным замыслом, прямое воплощение которого было неработоспособно! Такой возврат свидетельствует об очевидном прогрессе – в программе пользователя нет ничего лишнего!

Перейдем к реализации монитора:

```
with общий; use общий; — чтобы не переписывать
monitor буф is — !! в Аде такого нет, продолжаем писать на Ада-подобном ЯП
   entry поставить(X : in сообщение);
   entry получить(X : out сообщение);
end буф; — ключевые слова "monitor" и "entry" подчеркивают особую семантику
процедур "поставить" и "получить", то есть режим взаимного исключения

monitor body буф is
   procedure поставить(X: in сообщение) is
   begin
      if полон then ждать(неполон) end if;
      занести(X); — "обычная" запись в буфер
      послать(непуст); — сигнал для "получить"
   end поставить;
   procedure получить(X: out сообщение) is
   begin
      if пуст then ждать(непуст) end if;
      выбрать(X); — "обычная" выборка из буфера
      послать(неполон); — сигнал для "поставить"
   end получить;
end буф;
```

Чтобы все встало на свои места, в пакете «общий» нужно названия процедур «поставить» и «получить» заменить на «занести» и «выбрать» соответственно. Процедуры, объявленные в мониторе («мониторные», или «монопольные», процедуры), пользуются «обычными» пакетными процедурами, в которых можно конкретизировать такие особенности буфера, как его организация массивом или списком, очередность выборки сообщений и т. п., не относящиеся к параллелизму.

Важно понимать, что сам по себе режим взаимного исключения не спасает от тупиков. Ведь, например, при переполнении буфера процедура «поставить» не может нормально завершить свою работу, и если не внести уточнений в семантику монитора, то нельзя избежать тупика (пока не завершена процедура «поставить», не может работать «получить», чтобы освободить буфер!).

Поэтому на самом деле мониторные процедуры могут быть приостановлены в указанных программистом местах, с тем чтобы дать возможность запускать другие процедуры. Для этого можно воспользоваться аппаратом сигналов, что и сделано в нашем примере. Так что, например, первая процедура может приостановиться на операторе «ждать(неполон)», и так как в мониторе не остается активных процедур, он готов при необходимости активизировать вторую процедуру (которая в этом случае наверняка пошлет сигнал «неполон»), а после завершения второй процедуры сможет продолжить свою работу первая.

Итак, в семантику сигналов также внесена «мониторная» коррекция: можно возобновлять процесс из очереди только при условии, что он не приостановлен на процедуре из активного монитора.

Полезно подчеркнуть аналогию между взаимодействием мониторных процедур и сопрограмм.
Напомним, что такое процессы-сопрограммы X и Y:

```
process X;                process Y;
   ...                       ...
   resume Y;                 resume X;
   ...                       ...
   resume Y;                 resume X;
   ...                       ...
   detach;                   detach; => главная программа
```

Процессы-сопрограммы исполняются на одном процессоре. Оператор resume приостанавливает исполнение сопрограммы, в которой находится, и возобновляет исполнение указанной в нем сопрограммы с того места, на котором она была ранее приостановлена (или с самого начала, если это первое обращение к ней). Оператор detach возвращает управление главной программе.
Как и в случае сопрограмм, для каждого монитора предполагается единственный исполнитель, способный приостанавливать исполнение мониторных процедур (на операторе «ждать») и возобновлять их исполнение с прерванного места. Однако, в отличие от сопрограмм, приостановка не означает немедленного возобновления некоторой фиксированной «сопрограммы», указываемой оператором приостановки, – этому исполнителю приходится ждать явной активизации мониторной процедуры или появления нужного сигнала.

Сделаем выводы.

1. Мониторы обеспечивают высший уровень рациональной абстракции от особенностей параллелизма и удобные средства конкретизации, то есть это отличное средство структуризации программ.
2. Мониторы обеспечивают надежность предоставляемых процессам-пользователям услуг – пользователь не в силах «сломать» хорошо спроектированный и отлаженный монитор.
3. Мониторы обеспечивают ясность программирования процессов пользователей. Достаточно сравнить наш «идеал», в котором нет ничего лишнего, и решение с помощью семафоров.
4. Мониторы обеспечивают эффективность, когда их используют вместе со средствами пассивного ожидания (например, сигналами).
5. Режим взаимного исключения в мониторе встроен, синхронизация обеспечивается частично этим режимом, а в основном – посредством сигналов, развязка (независимость процессов) – асинхронностью процессов-пользователей и правилами приостановки мониторных процедур.

Итак, мониторы многим хороши, но:

- представляя собой средство высокого уровня (абстракции), требуют низкоуровневых средств для своей реализации (сигналов), то есть не могут служить единой концептуальной основой ЯПП;

- провоцируют создание административной иерархии программ там, где, по сути, достаточно их непосредственного (горизонтального) взаимодействия. В результате такую систему трудно перестраивать, если на один монитор приходится много процессов-клиентов (а если мало, то оказывается относительно много мониторов-посредников).

  Это, кстати, встроенные недостатки любой административной системы.

Имеются и другие причины (в частности, невозможность единой стратегии исполнения мониторных процедур, гарантирующей от тупиков), стимулирующие поиск иных языковых средств. Мы выделим в качестве важнейших две:

1) поиск единой концептуальной основы параллелизма, обеспечивающей приемлемые средства абстракции-конкретизации без обязательных дополнительных средств низкого уровня;
2) поиск выразительных средств, обеспечивающих «демократическое» взаимодействие процессов без лишних посредников.

Другими словами, требуется единая основа параллелизма, обладающая достаточно высоким уровнем, чтобы обеспечить ясность и надежность пользовательских процессов, и вместе с тем достаточно низким уровнем, чтобы было легко запрограммировать и семафоры, и сигналы, и мониторы, и другие полезные примитивы.

6.6. Рандеву

Основная идея (Хоар, Хансен – 1978 г.): соединить синхронизацию и обмен в одном примитиве, моделирующем встречу (свидание, **рандеву**) процессов-партнеров.

Например, для передачи данных из переменной X процесса A в переменную Y процесса B следует написать:

```
task A is
   X : сообщение;
   ...
   B ! X; -- заказ рандеву
   -- с процессом B для передачи
   -- из переменной X
   ...
end A;
```

```
task B is
   Y : сообщение;
   ...
   A ! Y; -- заказ рандеву
   -- с процессом A для приема
   -- в переменную Y
   ...
end B;
```

Семантику рандеву опишем на псевдокоде так:

```
if непуста (очередь партнеров) then
   [выбрать партнера из очереди; выполнить присваивание Y:=X;
   активизировать партнера];
else [приостановить текущий процесс и поместить его в очередь партнеров по рандеву
к процессу, с которым заказывается рандеву];
```

Итак, процесс A, желающий передать сообщение процессу B (своему партнеру), должен «заказать» с ним рандеву посредством конструкта B!X. Чтобы передача состоялась, процесс B должен также заказать рандеву посредством двойственного конструкта A?Y.

Внешняя среда принимает эти заказы и приостанавливает партнера, первым подавшего заказ (первым пришедшего на рандеву) до подачи заказа вторым партнером (то есть до его «прибытия» на рандеву). Когда оба партнера готовы, рандеву происходит «мгновенно» (с точки зрения партнеров, «счастливые часов не наблюдают»).

> Последнее сказано, скорее, для красного словца. Лучше было бы сказать, что рандеву начинается и заканчивается для партнеров одновременно (симметрично).

Итак, в рандеву соединены синхронизация (взаимное ожидание) и обмен (присваивание). В рандеву воплощена вполне «демократическая» идея «горизонтальных», прямых связей между партнерами. В результате общие переменные не нужны. Не нужен и режим взаимного исключения при доступе к ним. Однако все это только в случае единичного обмена.

При регулярном обмене, как в задаче «поставщик–потребитель», требуется еще и развязка партнеров, которую рандеву само по себе не обеспечивает – ведь темп обмена ограничен возможностями медленного партнера.

Эта проблема решается за счет моделирования посредством рандеву активного разделяемого ресурса – аналога пассивного буфера-монитора. Здесь существенно используется относительно низкий уровень такого примитива, как рандеву, – с его помощью легко моделировать нужные конструкты. Правда, для этого требуются процессы-посредники – и это основной недостаток рандеву как примитива.

Итак, наш новый (активный) буфер будет процессом-посредником, взаимодействующим посредством рандеву и с поставщиком, и с потребителем. Назовем этот процесс «буф»:

```
task Пост is                        task Потр is
   ...                                 ...
   буф ! X;                            буф ? Y;
   ...                                 ...
end поставщик;                      end потребитель;
```

Полезно обратить внимание на эту конфигурацию. Она может служить источником новых идей (см. ниже о каналах).

```
task буф is
   ...
   Пост ? Z;
   ...
   Потр ! Z;
   ...
end буф;
```

6.7. Проблемы рандеву

Итак:

a. Рандеву требует дополнительных процессов – ведь оно по замыслу связывает только активные объекты (при их взаимном «согласии»).

б. Достаточно взглянуть на наш «буф», чтобы понять, что без специальных средств отбора возможных рандеву невозможна развязка. Другими словами, нельзя менять темп рандеву с поставщиком относительно темпа рандеву с потребителем – нужно учитывать, к какому именно виду рандеву (из двух возможных) готовы потенциальные партнеры.
Поскольку о такой готовности знает только операционная (внешняя) среда, она и должна доставлять соответствующие средства отбора готовых к рандеву партнеров. Примеры таких средств рассмотрим позже (это, например, оператор select в Аде, alt в Оккаме).
Раньше проблемы отбора не возникало потому, что буфер был пассивным, – формально его готовность к работе не требовалась.

в. Партнеры должны называть друг друга по именам (должны «знать» друг друга). Это неудобно, если роль одного из партнеров – обслуживать произвольных клиентов. Примером может служить библиотечный пакет, которому вовсе не обязательно «знать» имена пользующихся его услугами процессов. По этому принципу устроен любой общедоступный сервис.
Конечно, имя партнера может быть параметром обслуживающего процесса (назовем его для краткости «мастером», в отличие от обслуживаемых процессов – «клиентов»). Но в таком случае возникают неприятные вопросы о способе передачи значения такого параметра. Статическая передача имени клиента (в период трансляции) не дает возможности менять клиентов в динамике. А динамическая передача требует либо рандеву с «неизвестным» клиентом, что невозможно, либо дополнительных «настраивающих» процессов, которым имена клиентов становятся известны не в результате рандеву.

Таким образом, **практически невозможно создание библиотечных мастеров посредством симметричного рандеву**. Итак, с одной стороны, доказана очередная неформальная теорема (о симметричном рандеву). С другой – именно она вынудила Бринча Хансена при разработке средств параллелизма в Аде отказаться от симметричного рандеву и ввести асимметричное, чтобы удовлетворить критичную для Ады потребность в библиотечных «мастерах».

Легко понять, почему для Ады проблема библиотечных мастеров критична – ведь это базовый ЯП для систем реального времени (то есть прежде всего для создания библиотечных пакетов, обслуживающих потребности программ реального времени). Важно и то, что асимметричное рандеву лучше согласуется с адовской концепцией строгого контроля типов.

6.8. Асимметричное рандеву

Основная идея: «сервис вместо свидания» – сохранив партнерство взаимодействующих процессов (оба партнера должны быть готовы взаимодействовать), свести собственно взаимодействие к исполнению некоторого аналога процедуры, определяемой только в одном из партнеров (обслуживающем партнере, «мастере»).

Иногда говорят «процесс-слуга» и «процесс-хозяин». Однако так менее выразительно. Ведь слуга знает хозяина (если только это не «слуга народа»). А здесь идея именно в том, чтобы обслуживающий процесс был пригоден и для анонимного клиента. Поэтому мы и предпочитаем термины «клиент» и «мастер».

Точнее говоря, для определенного вида взаимодействия (вида рандеву) выделяется процесс-мастер, предоставляющий услуги этого вида при соответствующем рандеву. Остальные процессы по отношению к этому виду услуг (рандеву) считаются клиентами, получающими услуги в моменты рандеву этого вида с соответствующим мастером.

При этом для клиента предоставление ему услуги неотличимо от вызова им подходящей процедуры. Другими словами, он совершенно «не замечает» асинхронного характера своего взаимодействия с мастером. Все особенности параллелизма (реального или виртуального) сказываются формально только на мастере. Это проявляется, в частности, в том, что в мастере предоставление услуги-рандеву оформляется специальными операторами (так называемыми операторами приема входа «accept» и др.).

Итак, при переходе к асимметричному рандеву:

а) можно написать библиотечного мастера;

б) в отличие от симметричного рандеву, проще реализовать произвольные услуги, а не только передачу значений переменных (**произвольную услугу неудобно разбивать между партнерами, обменивающимися значениями переменных, но вполне удобно запрограммировать аналогично некоторой процедуре**);

в) требуется, как и для симметричного рандеву, специальный аппарат управления (аналогично аппарату, обслуживающему ранее рассмотренные примитивы); в Аде это операторы accept, select и **объявление входа** (содержательно это объявление вида рандеву).

6.9. Управление асимметричным рандеву (семантика вспомогательных конструктов)

Объявление входа в Аде имеет вид заголовка процедуры, перед которым стоит ключевое слово entry. Например:

```
task семафор is
   entry оградить;     -- параметры не нужны (почему?)
   entry освободить;
end семафор;
```

Здесь записана спецификация задачи (в Аде), моделирующей семафор. Другими словами, она описывает процесс-мастер, предоставляющий такие услуги-рандеву, которые содержательно позволяют ему выступать в роли семафора (если снабдить его соответствующим телом (см. ниже)).

С точки зрения процесса-клиента, к входу мастера можно обращаться как к процедуре. Например:

```
...
оградить;
...
освободить;
...
```

Однако имеется существенное отличие от обычных процедур – процедуры-входы одного мастера работают в режиме взаимного исключения (это – следствие семантики рандеву), в то время как в общем случае в Аде процедуры считаются повторно-входимыми, а процедуры одного пакета могут активизироваться асинхронно (в том числе одновременно) из разных процессов.

Рассмотрим теперь оператор приема входа. Мастер считается готовым к рандеву, когда управление в нем достигает специального оператора «приема входа» вида

```
accept  < заголовок_процедуры >
   [ do < операторы > end ];
```

Заголовок_процедуры здесь совпадает с написанным после соответствующего entry в объявлении входа.

Когда и партнер-клиент готов (то есть его управление достигает оператора вызова соответствующей процедуры-входа), то требуемая синхронизация считается достигнутой, и происходит рандеву. Оно состоит в том, что после подстановки аргументов вызова исполняются операторы между do и end (если они есть). После этого рандеву считается состоявшимся, и партнеры вновь продолжают работать асинхронно.

Оператор отбора входов (в Аде это оператор select) необходим, как уже говорилось, для обеспечения развязки, чтобы рандеву разных видов не были жестко зависимы друг от друга. Его главное назначение – учет готовности клиентов к рандеву (и других условий), с тем чтобы не ждать рандеву с теми клиентами, которые «опаздывают» (не готовы к рандеву).

Общий вид этого оператора:

```
select
   [ when условие ==> ] отбираемая_альтернатива
      последовательность_операторов
   or
   ...
   or
   [when условие == > ] отбираемая_альтернатива
      последовательность_операторов
   [ else последовательность_операторов ]
end select;
```

Отбираемой альтернативой может быть оператор приема, оператор задержки или оператор завершения задачи. Когда управление в задаче достигает оператора отбора, то, во-первых, вычисляются все условия. Те альтернативы, для которых условие оказалось истинным, считаются «открытыми». Затем среди открытых альтернатив рассматриваются операторы приема, для которых очередь вызовов

соответствующих входов непуста. Если такие найдутся, то произвольным (с точки зрения программиста, но не создателя Ада-транслятора) образом выбирается одна из таких альтернатив и происходит соответствующее рандеву. Затем выполняется последовательность операторов, расположенная за этой отобранной альтернативой, и оператор отбора считается выполненным.

Если среди открытых альтернатив не оказалось операторов приема, готовых к рандеву, то выполняется оператор задержки (delay) на указанное количество секунд (если за это время возникает готовность к рандеву у открытых операторов приема, то отбирается альтернатива, готовая к рандеву, и оператор отбора завершается, как обычно). После задержки и выполнения соответствующей выбранной альтернативы (accept или delay) последовательности операторов оператор отбора входов считается выполненным.

Если одной из открытых альтернатив оказался оператор завершения (terminate), то (если нет готовых к рандеву операторов приема) при определенных дополнительных условиях задача может быть завершена (до этого должны, в частности, завершиться запущенные нашей задачей подчиненные задачи).

Альтернатива «иначе» (else) может быть выбрана, если нет открытых операторов приема, готовых к рандеву. Если в else стоит задержка, то во время этой задержки альтернативы уже не проверяются.

В одном операторе отбора, кроме операторов приема (хотя бы один оператор приема обязателен), допустимы либо только задержки, либо только завершение, либо только альтернатива «иначе».

В Аде имеются и другие разновидности оператора select, позволяющие не только мастеру не ждать не готового к рандеву клиента, но и клиенту не попадать в очередь к не готовому его обслужить мастеру.

6.10. Реализация семафоров, сигналов и мониторов посредством асимметричного рандеву

Продемонстрируем применение описанных средств управления рандеву на примерах моделирования посредством рандеву рассмотренных ранее примитивов.

Семафоры (спецификацию задачи «семафор» см. выше на стр. 167).

```
task body семафор is
begin
   loop
      accept оградить;    — только синхронизация
      accept освободить; — без обмена — нет части «do»
   end loop;
end семафор;
```

Видно, что из всех богатых возможностей рандеву используется только синхронизация – нет параметров и тела оператора приема. К тому же подчеркнута по-

следовательность операций «оградить – освободить», невозможность нарушить их порядок.

Сигналы

```
task сигнал is
   entry послать;
   entry ждать;
end сигнал;

task body сигнал is
      есть : boolean := false;
begin
      loop
         select
            accept послать; есть := true;
            -- присваивание – вне рандеву;
            -- во время рандеву ничего не делается!
         or
            when есть => accept ждать; есть := false;
         or
            delay t;
            -- задержка на фиксированное время t
         end select;
      end loop;
end сигнал;
```

Если нет открытых операторов приема, для которых клиенты готовы, то в данном случае оператор отбора будет t секунд ждать, не появятся ли клиенты. Если так и не появятся, считается выполненной последняя альтернатива, а вместе с ней – и весь оператор отбора. Затем – очередной цикл.

Видно, что сигнал применяется для связи разных процессов – потребовалась развязка, обеспечиваемая оператором отбора. В семафоре она была не нужна. Ведь сигнал, в отличие от семафора, не ждет на операторе приема. Он «всегда готов» обслужить любой процесс, но только по входу «послать».

Только что рассмотрен «незабываемый сигнал». Когда такой сигнал послан, то мастер о нем помнит до тех пор, пока его не воспримет процесс, ждущий этого сигнала. Возможна и иная интерпретация сигналов:

```
task body сигнал is – забываемый сигнал; спецификация та же, лишь тело другое
begin
   loop
      accept послать;
      select
         accept ждать;
         else
         null; – если сигнала не ждут, можно о нем забыть
      end select;
   end loop;
end сигнал;
```

Защищенные разделяемые переменные – мониторы

```
task защищенная_переменная is
   entry читать(X : out сообщение);
   entry писать(X : in сообщение);
end защищенная_переменная;

task body защищенная_переменная is
   Z : сообщение;
begin
   loop
      select
         accept читать(X : out сообщение) do X:=Z end;
      or
         accept писать(X : in сообщение) do Z:=X end;
      end select;
   end loop;
end защищенная_переменная;
```

В сущности, это монитор, реализующий режим взаимного исключения для процедур доступа к разделяемому ресурсу. Обратите внимание, что в полной мере обеспечены синхронизация и исключение, но качество развязки зависит от реализации языка (формально исполнитель имеет право отбирать, например, всегда первую альтернативу, даже если «второй» клиент давно ждет).

Лучше в этом смысле работает описанный ниже монитор «буф» (с дополнительными условиями отбора).

Монитор – буфер

```
with общий; use общий;
task буф is
   entry передать(X : in сообщение);
   entry получить(X : out сообщение);
end буф;
```

Как было! Пользоваться так же удобно и надежно.

```
task body буф is
begin
   loop
      select
         when not полон =>
            accept передать(X: in сообщение) do
               занести(X);
            end передать;
      or
         when not пуст =>
            accept получить(X : out сообщение) do
               выбрать(X);
            end получить;
      or
         delay t;
```

```
      end select;
   end loop;
end буф;
```

Итак, мы полностью смоделировали монитор Хансена-Хоара посредством рандеву. При этом семафоры не нужны, так как взаимное исключение обеспечивает select; сигналы не нужны благодаря проверке перед accept (рандеву вида «передать» просто не будет обслужено, пока функция «полон» вырабатывает логическое значение true). Причем эти проверки происходят в одном процессе буф, никаких проблем с прерываниями при таких проверках нет.

Таким образом, мы получили то, к чему стремились, – асимметричное рандеву может служить универсальным и надежным средством программирования параллельных процессов**.**

Вопрос. Зачем нужна альтернатива с задержкой?

Подсказка. Если нельзя выбрать ни одной альтернативы при операторе отбора, то возникает исключительная ситуация.

6.11. Управление асинхронными процессами в Аде

Рассмотрим (частично уже известные) сведения об асимметричном рандеву в рамках его воплощения в Аде.

Кроме основного примитива-рандеву, в ЯП нужен аппарат **управления рандеву** (сравните операторы «оградить», «освободить», «послать», «ждать» для ранее рассмотренных примитивов). В Аде аппарат управления рандеву состоит из ОБЪЯВЛЕНИЯ ВХОДА (entry), ОПЕРАТОРА ВЫЗОВА ВХОДА (синтаксически не отличимого от вызова процедуры), оператора ПРИЕМА (accept), оператора ОТБОРА ВХОДОВ (select) и некоторых других.

Подчеркнем, что процедуры в Аде все считаются повторно-входимыми (к ним можно независимо обращаться из различных асинхронных процессов, и их тела могут одновременно исполняться в этих процессах). Входы отличаются от процедур, во-первых, тем, что обращения к ним из различных процессов выполняются строго в порядке очереди (именно здесь встроено взаимное исключение процессов-конкурентов), во-вторых, наличием не одного, а многих «тел» – операторов приема, расположенных и исполняемых в различных точках процесса-мастера.

Оператор ЗАДЕРЖКИ (delay) приостанавливает исполнение задачи, в которой он находится, на указанный в нем период (реального, астрономического) времени.

Вызов ВХОДА R, находящийся в задаче K, аналогичен вызову процедуры, но в общем случае не исполняется немедленно, а лишь «заказывает РАНДЕВУ» категории R. Это значит, что задача K (клиент по входу R) готова к рандеву с другой задачей-мастером M, в которой вход R объявлен. Она оказывается готовой обслужить заказ задачи K лишь тогда, когда достигнет оператора ПРИЕМА (accept)

входа R. Оператор приема предписывает действия, выполняемые в момент рандеву. Когда эти действия завершаются, рандеву считается состоявшимся, и обе задачи могут продолжать асинхронно работать (до следующего взаимодействия-рандеву). Если задача M достигает оператора приема входа R раньше, чем его закажет какая-либо обслуживаемая задача, то задача M приостанавливается и ждет появления заказов (ждет рандеву).

Таким образом, рандеву происходит тогда (и только тогда), когда и клиент, и мастер оказываются к нему готовыми (задача K дошла до вызова входа и заказала рандеву категории R, а задача M дошла до оператора приема и готова выполнить заказ).

Собственно рандеву состоит в том, что аргументы вызова входа R (из задачи-клиента) связываются с параметрами оператора приема (из задачи-мастера) и выполняется тело оператора приема.

Все происходит так, как будто из задачи K обращаются к процедуре R, объявленной в задаче M. Выполнение оператора приема в задаче M означает тем самым и выполнение оператора вызова в задаче K (и тем самым завершение рандеву категории R). Другими словами, задачи K и M как бы сливаются на время рандеву, а затем продолжают работать независимо до следующего возможного рандеву.

Оператор ОТБОРА ВХОДОВ (select) позволяет мастеру ожидать сразу нескольких рандеву и отбирать (из заказанных!) те рандеву, которые удовлетворяют указанным в этом операторе УСЛОВИЯМ ОТБОРА.

Формально спецификация задачи – это объявление объекта анонимного задачного типа. Оно связывает имя задачи с объектом, который представляет асинхронный процесс, определяемый телом соответствующей задачи. Таким образом, данные задачных типов – активные данные. Объект задачного типа, то есть асинхронный процесс, запускается (начинает работать) в момент, когда заканчивается обработка объявления объекта (в нашем случае – объявления задачи).

В языке Ада можно объявить именованный задачный тип. Например:

```
task type анализ is
   entry прими(X: in сообщение );
end анализ;
```

Такое объявление связывает имя «анализ» с задачным типом, класс значений которого – асинхронные процессы с входом «прими», определяемые телом задачи с именем «анализ».

В контексте, где доступен задачный тип «анализ», можно объявить индивидуальную задачу этого типа, например:

```
A: анализ; — то есть обычное объявление объекта.
```

При обработке такого объявления запускается новый асинхронный процесс типа «анализ» (то есть создается новый объект задачного типа «анализ»), и с ним связывается имя A. Если нужно, можно запустить и другие процессы этого типа объявлениями

```
A1: анализ;
A2: анализ; — и т. д.
```

При этом доступ к входу «прими» нужного процесса обеспечивает составное имя вида A1.прими, A2.прими и т. п.

Когда же имеется единственный процесс с входом «прими», имя задачи можно не указывать и пользоваться простым именем входа.

Обратите внимание, что входы задач можно рассматривать как аналоги селекторов в объектах комбинированных типов. Обращение к входу по составному имени напоминает выборку значений поля. В определяющем пакете для задачного типа могут быть объявлены подходящие базовые операции. Все сказанное и позволяет считать задачные типы полноценными (ограниченными!) типами данных, причем данных активных, а не пассивных.

Объекты задачных типов могут служить компонентами объектов составных типов. Например, можно объявить массив из десяти анализаторов:

```
A: array (1..10) of анализ;
```

и обращаться к соответствующим входам с помощью индексации

```
A(1).прими ...; ...; A(10).прими ...
```

Задачные объекты могут, естественно, быть и динамическими. Например, можно ввести ссылочный тип

```
type P is access анализ;
```

и переменную R типа P

```
R : P;
```

Теперь понятно действие оператора

```
R := new анализ;
```

А именно создается новый асинхронный процесс типа «анализ» и ссылка на него помещается в R. К соответствующему входу можно теперь обращаться через

```
R.прими ...
```

Подчеркнем, что у динамических задачных объектов не может быть динамических параметров, так что все сказанное про соответствие задачных типов концепции уникальности сохраняет силу и для динамических задачных объектов.

Формально тело задачи отличается от тела процедуры лишь тем, что в первом допустимы специальные «задачные» операторы, недопустимые в теле обычной процедуры.

Как уже сказано, рациональной структуризацией управления асинхронными процессами много и плодотворно занимался Бринч-Хансен. Интересующегося читателя отсылаем к [13]. Практическим результатом исследований проблем параллелизма еще одним классиком информатики Тони Хоаром стал язык параллельного программирования Оккам. Ему (точнее, его последней версии Оккам-2) посвящен специальный раздел.

Глава 7

Нотация

7.1. Проблема знака в ЯП 176
7.2. Определяющая потребность 176
7.3. Основная абстракция 177
7.4. Проблема конкретизации эталонного текста 177
7.5. Стандартизация алфавита 178
7.6. Основное подмножество алфавита 179
7.7. Алфавит языка Ада 179
7.8. Лексемы 180
7.9. Лексемы в Аде 181

7.1. Проблема знака в ЯП

Вспомним, что ЯП – знаковая система. Но знаковая ситуация возникает лишь тогда, когда знак может быть передан отправителем и получен адресатом. Как отправителем, так и адресатом может оказаться и человек, и компьютер.

ЯП должен быть средством мышления людей, создающих программы; средством их общения между собой по поводу создания программ; средством общения людей с компьютерами и, наконец, компьютеров между собой.

Для людей важно, чтобы знаки были и выразительны, и надежны, и лаконичны, и удобны для письма и чтения. Необходимость общаться с компьютерами предъявляет к знакам особые требования. Достаточно вспомнить, что знаки ЯП нужно вводить устройствами ввода и выводить устройствами вывода. К тому же они должны восприниматься имеющейся в распоряжении программной средой.

Указанные требования весьма разнообразны и порой противоречивы. Привычных людям знаков часто нет на устройствах ввода-вывода. Может оказаться невозможным использовать буквы кириллицы, некоторые привычные символы операций, опускать и поднимать индексы и т. п. Обычно невозможно вводить рукописный текст (хотя в будущем это наверняка станет возможным).

Итак, даже если в знаковой ситуации, соответствующей ЯП, сконцентрировать внимание исключительно на выборе знаков, по возможности абстрагируясь от проблемы смысла, то найти решение, удовлетворяющее в разумной мере пользователей ЯП и производителей оборудования, бывает очень не просто. Когда же необходимо искать решение с учетом массового применения ЯП в национальном (тем более мировом) масштабе, то возникает самостоятельная серьезная проблема – проблема знака.

В этом разделе мы сконцентрируемся лишь на части этой большой проблемы – проблеме представления знаков (проблеме нотации).

7.2. Определяющая потребность

Выделим технологическую потребность, определяющую в настоящее время решение проблемы нотации, – потребность записывать программу так, чтобы ее можно было ввести в любой компьютер без особых затрат и риска внести ошибки. Назовем ее потребностью совместимости по вводу. Эта потребность – определяющая в том смысле, что ради ее удовлетворения в современной ситуации с индустриальным программированием можно в значительной степени пренебречь, например, пожеланиями некоторых категорий пользователей (разнообразие шрифтов, управление цветом, нелинейная запись и т. п.).

Другими словами, пишущий программу (отправитель знака) должен иметь возможность абстрагироваться от особенностей устройств ввода у адресата. С другой стороны, нужно обеспечить возможность «каждому желающему» конкретному исполнителю выступить в роли адресата, возможность воспринять написанное отправителем.

7.3. Основная абстракция

Абстракция, обслуживающая потребность совместимости по вводу, хорошо известна – это абстрактный (эталонный) текст. Понятие эталонного текста определено в каждом ЯП. Эталонный текст – это конечная последовательность эталонных символов. Набор символов в ЯП обычно конечен и линейно упорядочен. Он называется алфавитом ЯП. Потребность в совместимости удовлетворяется за счет того, что на определенном уровне абстракции именно эталонный текст является знаком программы. Именно он подразумевается, когда работают на конкретном устройстве ввода-вывода.

Но на конкретном устройстве свой алфавит. Так что приходится придумывать способ обозначать эталонные символы конкретными символами, доступными на устройстве, а эталонный текст в целом – конкретным текстом (составленным из конкретных символов). Так, эталонные символы Алгола-60 (begin, end и т. п.) обозначаются иногда "BEGIN", "END", иногда _begin_ , _end_ , иногда 'НАЧАЛО', 'КОНЕЦ' и т. п.

Таким образом, конкретный текст обозначает эталонный, а тот, в свою очередь, обозначает программу.

Итак, основная абстракция осознана – это эталонный текст. Но в соответствии с принципом реальности абстракций для каждой абстракции нужны средства конкретизации. Проблема нотации дает пример, когда средства конкретизации по необходимости выходят за рамки языка, создавая внешнюю проблему конкретизации эталонного текста.

7.4. Проблема конкретизации эталонного текста

Обычно в ЯП отсутствуют средства управления связью конкретных и абстрактных текстов. Дело в том, что средства управления сами должны быть обозначены некоторыми текстами, их также нужно вводить и выводить. Короче, для них возникнут те же проблемы, что и для ЯП в целом. Так что решать проблему конкретизации приходится вне ЯП.

Тем более важно принять рациональные решения, определяющие правила конкретизации абстрактных текстов, так как они принимаются «раз и навсегда».

Важность решений, о которых идет речь, можно показать на классическом примере Алгола-60. В свое время его авторы по существу игнорировали проблему конкретизации. Они ввели три уровня языка – эталонный, для публикаций и конкретные представления. Первый был ориентирован «исключительно на взаимопонимание», второй – на «типографские особенности», третий – на устройства ввода-вывода. Что касается проблемы конкретизации, то авторы ограничились оговоркой, что каждая реализация должна иметь «правила для перевода конкретных представлений в эталонные».

Именно «правила», а не программы и не программные изделия для перевода! Затраты ресурсов на такой перевод и риск внести ошибки не оценивались и не контролировались. На практике это привело к несовместимости различных трансляторов с Алгола-60. Так как реализаторы не только не подкрепляли «правила» конкретными программными изделиями, но и не всегда четко осознавали «правила». Проблема несовместимости реализаций, в свою очередь, сыграла не последнюю роль в том, что Алгол-60 не сумел выдержать конкуренцию с Фортраном в качестве языка массового программирования для научных расчетов.

Итак, допустим, что важность проблемы конкретизации осознана. Как рационально решить эту проблему?

7.5. Стандартизация алфавита

Игнорировать проблему нельзя, управлять конкретизацией невозможно. Остается по существу единственный путь – стандартизация алфавита (или определение ЯП со стандартным алфавитом).

Ключевая идея состоит в том, что проблема выносится за рамки рассматриваемого ЯП и выбирается опорный стандарт на цифровые коды символов (достаточно распространенный, лучше всего – международный). Эталонный алфавит ЯП определяется через опорный стандарт (по существу, эталонным алфавитом становится некоторое подмножество цифровых кодов символов из опорного стандарта, а связанные с этими кодами видимые (графические) и (или) управляющие символы образуют допустимые конкретные алфавиты). Тем самым определяются и допустимые вариации конкретных алфавитов (рамками того же опорного стандарта).

С одной стороны, авторы ЯП вынуждены выбирать из стандартного набора символов. С другой стороны, производители оборудования и систем программирования вынуждены считаться с действующими стандартами и обеспечивать, во-первых, наличие на клавиатуре устройств минимального набора знаков и, во-вторых, их правильное, определяемое стандартом соответствие цифровым кодам (например, A – 101, B – 102, 0 (нуль) – 60, 1 – 61 в коде ASCII и т. п.). Таким образом, на некотором этапе обработки текст обязательно представлен стандартной последовательностью числовых кодов. Ее и следует считать эталонным текстом. Именно такой эталонный текст обеспечивает практическую совместимость по вводу.

Стандартизация алфавита требует коллективных усилий международного сообщества, самоограничения и дисциплины авторов ЯП, производителей компьютеров и периферийных устройств. Но зато и уровень совместимости по вводу в ЯП со стандартизированным алфавитом зависит не от распространенности конкретной реализации языка, а от распространенности опорного стандарта.

Благодаря целенаправленной деятельности национальных и международных организаций по стандартизации в настоящее время существуют достаточно авторитетные стандарты на символы (7- и 8-битовый Международные стандарты ИСО и соответствующие национальные стандарты, в том числе и отечественный

ГОСТ). Так что создана приемлемая база для разработки ЯП со стандартным алфавитом.

Рост технических возможностей и соответственно потребностей пользователей может привести к пересмотру стандартов на коды символов (например, чтобы можно было работать с цветом или с различными шрифтами). Тогда появится больше возможностей и у авторов ЯП. Вместе с тем потери от несовместимости обычно несопоставимы с выигрышем от нарушения стандарта, так что известная доля консерватизма в решении проблемы нотации вполне естественна.

Для ЯП со стандартным алфавитом нет особого смысла различать эталонные и конкретные тексты. Другими словами, абстракция представления в этом случае почти вырождается в результате стандартизации конкретных представлений.

Первым ЯП со стандартным алфавитом был Фортран. В настоящее время этот путь решения проблемы представления знака для вновь создаваемых ЯП можно считать общепринятым.

7.6. Основное подмножество алфавита

Еще одна заслуживающая внимания идея (позволяющая работать на «бедных» устройствах, не соответствующих полному опорному стандарту на коды символов) состоит в выделении так называемого основного подмножества алфавита. При этом в определении ЯП фиксируются правила изображения остальных символов алфавита с помощью комбинаций символов из основного подмножества. Написанный по этим правилам текст обозначает нужный текст в полном алфавите, а передавать и воспринимать его можно на «бедных» устройствах.

В любой «богатой» реализации ЯП можно (и нужно) иметь средства для кодирования и декодирования «бедных» текстов по упомянутым правилам, так что идея основного подмножества практически не мешает «богатым» пользователям и существенно помогает «бедным».

7.7. Алфавит языка Ада

Текст исходной программы в Аде – это последовательность символов. Символы делятся на графические и управляющие. Каждому символу однозначно соответствует 7-битовый код ИСО. Вариации графических символов возможны только в рамках, допустимых стандартом ИСО для национальных стандартов (например, знак доллара можно заменить знаком фунта стерлингов). Управляющие символы графического представления не имеют, они предназначены для форматирования текста (горизонтальная табуляция, вертикальная табуляция, возврат каретки, перевод строки, перевод страницы).

Среди графических символов выделено основное множество (прописные латинские буквы, цифры, пробел и специальные символы # &’()* + ,– .:;<=>_|.

Кроме того, в алфавит входят строчные латинские буквы и дополнительные символы (! $ % ? @ [\ ] ' '{}^).

Правила, позволяющие обозначить произвольную программу с помощью только основного множества, таковы. Во-первых, в качестве обязательных элементов программы (ключевые слова, ограничители и разделители) используются только символы из основного множества. Во-вторых, строчные и прописные буквы эквивалентны всюду, кроме строк и символьных констант. (Так что и идентификаторы можно представлять в основном множестве.) А строки обозначаются с помощью символа & так, что «явное» изображение строки эквивалентно «косвенному», использующему название нужной подстроки. Например, если ASCII.DOLLAR – это название строки «$», то обозначение «A $ C» эквивалентно «A» & ASCII.DOLLAR & «C».

Подобные названия для всех дополнительных символов и строчных латинских букв предопределены в языке Ада. Это и позволяет записать любую программу с помощью одного только основного множества. (Еще пример: «AвC» эквивалентно «A» & ASCII.LC_B & «C»; здесь LC служит сокращением от английского LOWER_CASE_LETTER – строчные буквы).

7.8. Лексемы

Понятие эталонного текста как последовательности символов (литер) позволяет абстрагироваться от особенностей устройств ввода-вывода. Однако символ – слишком мелкая единица с точки зрения тех сущностей, которые необходимо обозначать в ЯП. Их намного больше, чем элементов в алфавите. Удобно, когда эти сущности имеют индивидуальные обозначения, подобные словам естественного языка, а текст оказывается последовательностью таких «слов», называемых лексемами. Мы пришли к еще одному (промежуточному) уровню абстракции – уровню лексем. (Можно считать, что этот уровень удовлетворяет потребность в рациональной микроструктуре текста – приближает размеры «неделимого» знака к размеру «неделимого» денотата.)

Когда этот уровень абстракции выделен явно, и при письме, и при чтении можно оперировать достаточно крупными единицами (лексемами), абстрагируясь (когда это нужно) от конкретного способа представления лексем символами алфавита. Становится проще манипулировать с текстом, увеличивается надежность, растет скорость создания и восприятия текста.

Между тем в ЯП уровень лексем выделяется далеко не всегда. Неудачная идея игнорировать пробелы как естественные разделители возникла на заре развития ЯП (сравните Фортран и Алгол-60), по-видимому, как отрицательная реакция на необходимость «считать пробелы» в первых позиционных автокодах. В результате была временно утеряна отлично зарекомендовавшая себя традиция естественных языков – выделять слова пробелами. В Алголе-60 к тому же игнорируются все управляющие символы, а в Фортране – переход на новую строку внутри оператора. В естественных языках подобные особенности текста обычно используются как разделители слов. В последние годы идея явного выделения уровня лексем становится общепризнанной и при конструировании ЯП.

Интересно отметить, что «возвращение пробела» как значащего символа связано с пониманием «ключевых слов» просто как зарезервированных слов (а не «иероглифов», как в Алголе), ничем другим от остальных слов-лексем не отличающихся. Но тогда естественно запретить сокращать ключевые слова (иначе их можно спутать теперь уже не только с другими ключевыми словами, но и с идентификаторами). Это в целом полезное ограничение, так как способствует надежности программирования, помогая чтению за счет некоторой дисциплины письма (что вполне в духе индустриального программирования). Кстати, не очевидно, что напечатать слово procedure труднее, чем «proc», с учетом переключения внимания на спецзнаки. К тому же современные системы подготовки текстов позволяют легко вводить словари сокращений (так что и чтения не затрудняют, и печатать удобно).

7.9. Лексемы в Аде

Лексемы в Аде аналогичны словам естественного языка. Они делятся на шесть классов: ограничители (знаки препинания), идентификаторы (среди которых – зарезервированные ключевые слова), числа, обозначения символов, строки и примечания. В некоторых случаях, когда невозможно иначе однозначно выделить лексему, требуется явный разделитель между смежными лексемами. В качестве разделителя выступает или пробел, или управляющий символ, или конец строчки. Пробел, естественно, не действует как разделитель в строках, примечаниях и в обозначении пробела (‘ ‘). Управляющие символы (кроме, возможно, горизонтальной табуляции, эквивалентной нескольким пробелам) всегда служат разделителями лексем. Между лексемами (а также до первой и после последней лексемы) текста допустимы несколько разделителей. Заметим, что каждая лексема должна располагаться на одной строке (ведь конец строки – разделитель).

Со списком ключевых слов Ады мы познакомились по ходу изложения. Многие из них стали фактически стандартными для многих ЯП (procedure, begin, do и т. д.). Сокращать ключевые слова недопустимо.

Ниже следует описание классов лексем.

Ограничитель. Это одиночный символ

```
&'()*+,-./:;<=>
```

и пара символов

```
=>   ..   **   :=   /=   >=   <=   <<   >>   <>
```

При этом символ может играть роль ограничителя только тогда, когда он не входит в более длинную лексему (парный ограничитель, примечание, строку).

Идентификатор. Отличается от алгольного или паскалевского идентификатора только тем, что внутри него допускается одиночное подчеркивание. Прописные и строчные буквы считаются эквивалентными. Идентификаторы считаются различными, если отличаются хотя бы одним символом (в том числе и подчеркиванием), например 'A', ‘*’, '"', ' ' и т. п.

Строка. Это последовательность графических символов, взятая в двойные кавычки. Внутри строки двойная кавычка изображается повторением двойной кавычки («»), например «Message of the day».

Примечание. Начинается двумя минусами и завершается концом строки.

Число. Примеры целых чисел:

```
65_536 ,   10.000
2#1111_1111#   , 16#FF# ,  016#0FF#
-- целые константы, равные 255

16#E#E1 , 2#1110_0000#
-- это 222
```

Примеры вещественных чисел:

```
16#F.FF#E+2 ,     2#1.1111_1111_111#E11
-- 4095.0
(Пробелы внутри не допускаются — ведь они разделители.)
```

Глава 8

Исключения

8.1. Основная абстракция 184
8.2. Определяющие требования 185
8.3. Аппарат исключений в ЯП 187
8.4. Дополнительные особенности обработки исключений 194

8.1. Основная абстракция

Представим себе заводского технолога, планирующего последовательность операций по изготовлению, например, блока цилиндров двигателя внутреннего сгорания. Аналогия с программированием очевидна. Соответствующая технологическая карта (программа) предусматривает отливку заготовки, фрезеровку поверхностей и расточку отверстий. Каждый из этапов довольно подробно расписывается в технологической карте. Однако все предусматриваемые технологом подробности касаются создания именно блока цилиндров. В технологической карте, конечно, не сказано, что должен делать фрезеровщик, если выйдет из строя фреза, если возникнет пожар, землетрясение, нападет противник. Если бы технолог был вынужден планировать поведение исполнителя в любых ситуациях, то он никогда не закончил бы работу над такими «технологическими картами».

В сущности, специализация в человеческой деятельности основана на способности выделить небольшой класс ситуаций, считающихся существенными для этого вида деятельности, а от всех остальных абстрагироваться, считать чрезвычайными, необычными, исключительными, требующими переключения в другой режим, в другую сферу деятельности.

Примерам нет числа. Кулинарный рецепт не описывает поведения хозяйки, если в процессе приготовления блюда зазвонит телефон; физические модели применимы при определенных ограничениях и ничего не говорят о том, что будет при нарушении этих ограничений (например, из законов Ньютона нельзя узнать о поведении объектов при релятивистских скоростях).

Вместе с тем важно понимать, что теми аспектами планируемой деятельности, от которых приходится абстрагироваться, ни в коем случае нельзя пренебрегать. При реальном возникновении чрезвычайных обстоятельств именно они и становятся определяющими. Так что в жизнеспособной системе, а тем более системе, претендующей на повышенную надежность, совершенно необходим аппарат, обеспечивающий адекватную реакцию системы на чрезвычайные ситуации. К счастью, технолог обычно вправе рассчитывать на интеллект, жизненный опыт и общую квалификацию исполнителя-человека.

Программист, вынужденный создавать программу для автомата, по необходимости попадает в положение заводского технолога, которого заставляют писать инструкции по гражданской обороне или поведению во время пожара. Ведь надежная программа должна вести себя разумно в любых ситуациях. Как выйти из положения, нам уже нетрудно догадаться – снова абстракция (и затем конкретизация). Нужно иметь возможность, занимаясь содержательной функцией программы, отвлекаться от проблемы чрезвычайных обстоятельств, а занимаясь чрезвычайными обстоятельствами, в значительной степени отвлекаться от содержательной функции программы. Вместе с тем на подходящем этапе программирования и исполнения программы нужно, конечно, иметь возможность учесть все тонкости конкретных обстоятельств.

Таким образом, мы приходим к одной из важнейших абстракций программирования – абстракции от чрезвычайных обстоятельств, от особых (исключитель-

ных) ситуаций. Будем называть их для краткости просто «исключениями», а соответствующую абстракцию – абстракцией от исключений.

Проще говоря, речь идет об аппарате, поддерживающем систематическое разделение нормальной и ненормальной работы, причем не только программы, но в некотором смысле и программиста.

8.2. Определяющие требования

Требования к аппарату исключений в ЯП легко выводятся из самых общих соображений (и тем не менее нигде не сформулированы). Представим их в виде трех принципов, непосредственно следующих из назначения исключений в условиях систематического, надежного и эффективного программирования.

Принцип полноты исключений: на любое исключение должна (!) быть предусмотрена вполне определенная реакция исполнителя.

Формулировка не претендует на строгость. Конечно, имеется в виду «любое» исключение не из реального мира, а из «мира» исполнителя (трудно предусмотреть реакцию компьютера, например, на любую попытку его поломать). Но зато действительно имеется в виду любое исключение из этого «мира», что немедленно приводит еще к одному важному понятию, которому до сих пор мы не имели случая уделить достойного внимания.

Дело в том, что предусматривать определенную реакцию на любое исключение в каждой программе во всех отношениях неразумно. Читатель легко поймет, почему. Поэтому наиболее общие правила такого реагирования на исключения должны быть предусмотрены априори, еще до начала программирования, то есть авторами ЯП. Другими словами, существенная часть реакции на исключения должна предусматриваться «априорными правилами поведения исполнителя», а не собственно программой.

К сожалению, этот принцип в полной мере воплотить не удается по многим причинам (в частности, из-за невозможности абсолютно точно определить ЯП или безошибочно программировать). Однако он полезен в качестве ориентира для авторов ЯП и программистов.

Об априорных правилах поведения исполнителя

Планировать поведение исполнителя означает, в частности, согласовывать его модель мира с моделью решаемой задачи. Чем лучше эти модели согласованы априори, тем проще планировать поведение, проще достичь взаимопонимания. В сущности, изученные нами абстракции включают в ЯП именно для того, чтобы приблизить мир исполнителя к миру решаемых задач. Однако до сих пор у нас не было случая обратить внимание на то, что «мир исполнителя» не следует ограничивать операциями и данными (возможно, весьма мощными и разнообразными).

Сколь мощной ни была бы операция, ее нужно явно указать в программе. Между тем одна из самых общих технологических потребностей – потребность писать надежные программы, только что привела нас к абстракции новой категории, к абстракции от конкретной программы или понятию об априорных правилах поведения исполнителя (то есть правилах поведения, предопределенных в ЯП и не обязательно связанных непосредственно с какими-либо указаниями в программе).

Именно априорные правила поведения, а не специфические данные и операции характеризуют современные развитые ЯП. С этой точки зрения удобно рассматривать и особенности взаимодействия асинхронных процессов, и перебор с возвратом (backtracking), и вызов процедур по образцу, и поведение экспертных систем, и вообще программирование достаточно высокоинтеллектуальных исполнителей (в том числе программирование человеческой деятельности).

Связь с этим аспектом мира исполнителя вступает в действие при исполнении любой программы, как только возникает исключительная ситуация. Вместе с тем в развитом ЯП всегда имеются и специальные средства конкретизации, позволяющие корректировать априорные правила поведения с учетом потребностей конкретной программы.

Важно понимать, что без априорных правил поведения исполнителя не обойтись. Ведь если программист захочет все предусмотреть, он должен планировать проверку соответствующих условий. Но исключительные ситуации могут возникнуть в процессе исполнения программы проверки! Так что чисто программным путем задачу не решить. Необходим определенный исходный (априорный) уровень подготовки исполнителя.

В жизни примерами «общих правил» служат обязанность вызвать пожарную команду по телефону 01, милицию – по 02, скорую помощь – по 03, а также выполнить инструкцию по оказанию неотложной помощи пострадавшим, мобилизационное предписание и т. п.

Пример конкретизации – подробный план эвакуации людей из помещения при пожаре, план мобилизационных и эвакуационных мероприятий на случай войны, перечень занятий дежурного при отсутствии телефонных звонков (ведь для него обычная работа – отвечать на телефонные звонки, а их отсутствие – исключение).

Принцип минимальных возмущений: затраты на учет чрезвычайных обстоятельств должны быть по возможности минимальными (при гарантии сохранения жизнеспособности системы).

Этот же принцип в терминах ЯП: языковые средства должны быть такими, чтобы забота об исключениях в минимально возможной степени сказывалась на всех этапах жизненного цикла программных сегментов, реализующих основную содержательную функцию программы.

Другими словами, аппарат исключений в ЯП должен быть таким, чтобы программирование поведения в чрезвычайных обстоятельствах могло быть в максимально возможной степени отделено от программирования основной функции (в частности, не мешало понимать содержательную функцию программы), накладные расходы на исполнение основной функции могли быть минимальными и т. п.

Принцип минимальных повреждений: ущерб при возникновении исключений должен быть минимальным. Речь идет уже не о минимизации затрат на саму способность реагировать на исключения, а об ущербе (иногда принципиально неустранимом), который может быть нанесен при реальном возникновении чрезвычайных для программы обстоятельств.

Например, разумный аппарат исключений должен предусматривать возможность реагировать на исключение как можно раньше и как можно точнее, чтобы предотвратить дальнейшее разрушение программ и данных при аварийном их функционировании. Ясно, что этот принцип имеет смысл только тогда, когда ис-

ключения трактуются как аварии или реально оказываются таковыми (часто это как раз априорные исключения). Например, завершение файла при чтении часто удобно считать исключением с точки зрения «нормальной» обработки его записей, однако в этом случае не имеет смысла говорить об авариях или повреждениях.

Полезные примеры, помогающие лучше прочувствовать суть изложенных принципов, содержатся в [14].

8.3. Аппарат исключений в ЯП

Концепция исключения в ЯП содержательно имеет много общего с концепцией аппаратного внутреннего прерывания, однако могут быть и существенные отличия. Ближе всего к понятию прерывания трактовка исключений в языке ПЛ/1.

Выделим четыре аспекта аппарата исключений:

- определение исключений (предопределенные и определяемые);
- возникновение исключений (самопроизвольное и управляемое);
- распространение исключений (статика или динамика);
- реакция на исключения (пластырь или катапульта – см. ниже).

Кроме этого, уделим внимание другим особенностям исключений, в частности особенностям исключений в асинхронных процессах.

8.3.1. Определение исключений

Рассмотрим концепцию исключения, ориентируясь на Аду, стараясь больше уделять внимания «авторской позиции», то есть объяснять, почему при проектировании ЯП были приняты излагаемые решения. Основой послужат, конечно, принципы полноты, минимальных возмущений и минимального ущерба.

Все потенциальные исключения в программе на Аде имеют индивидуальные имена и известны статически. Они либо предопределены, либо объявлены (определены) программистом.

Предопределенные исключения касаются, естественно, самых общих ситуаций. Например, при нарушении ограничений, связанных с типом (ограничений допустимого диапазона значений, диапазона индексов и т. п.), возникает (иногда говорят «возбуждается») исключение нарушение_ограничения (constraint_error); при ошибках в числовых расчетах (переполнение, деление на нуль, исчезновение и т. п.) – исключение численная_ошибка (numeric_error); при неправильной компоновке программы (отсутствие тела нужного программного сегмента и т. п.) – исключение нет_сегмента (program_error); при нехватке памяти для размещения динамических объектов – исключение нет_памяти; при нарушении во взаимодействии асинхронных процессов (аварийное или нормальное завершение процесса, содержащего вызываемый вход, и т. п.) – исключение ошибка_взаимодействия (tasking_error).

Если, например, объявить

```
A:array(1 .. 10) of INTEGER;
```

то при I=11 или I = 0 в момент вычисления выражения A(I) возникает предопределенное исключение нарушение_ограничения.

На уровне ЯП принцип полноты исключений обеспечен тем, что нарушение любого языкового требования при выполнении программы на Аде приводит к возникновению некоторого предопределенного исключения.

Принцип минимальных возмущений проявляется в том, что предопределенные исключения возникают без какого бы то ни было указания программиста. Так что, во-первых, программист избавлен от необходимости проектировать их возникновение, во-вторых, упомянутые указания не загромождают программу и, в-третьих, соответствующие предопределенные проверки могут быть в принципе реализованы аппаратурой или авторами компилятора так, чтобы требовать минимума ресурсов при исполнении программы. Например, во фрагменте программы

```
<<B>>
declare
   A: float;
begin
   ...
   A:=X*X;
   Y:=A*EXP(A);    -- здесь возможно переполнение при возведении в степень или
                   -- умножении
   ...
exception — ловушка исключений
   when NUMERIC_ERROR => Y:=FLOAT'LAST; — наибольшее вещественное
   PUT('Переполнение при вычислении Y в блоке B');
end B;
```

четко отделена часть, реализующая основную функцию фрагмента (до ключевого слова exception), от части, реализующей предусмотренную программистом реакцию на предопределенное исключение численная_ошибка, возникающее при переполнении, – это ловушка исключений.

Первую часть программист мог писать, не думая об исключениях вообще, а затем мог добавить ловушку. Работать эта ловушка будет тогда и только тогда, когда возникнет исключение, а затрат на проверку переполнения в программе нет вовсе – обычно это дело аппаратуры. После реакции на исключение переменная Y получит «правдоподобное» значение, позволяющее сохранить работоспособность программы после аварии, а программист получит точное сообщение об аварии в своих собственных обозначениях (размещенное в операторе PUT).

> **Упражнение.** Попытайтесь написать эквивалентную программу, не пользуясь аппаратом исключений. Убедитесь, что при этом приходится нарушать все три принципа со стр. 185–186.

Принцип минимальных повреждений (минимального ущерба) в современных ЯП почти не влияет на возникновение исключений (хотя мог бы и влиять, позволив управлять информацией, которая становится доступной при возникновении исключения). Зато он существенно влияет на их определение и обработку. В част-

ности, именно для того, чтобы программист мог предусмотреть быструю и точную реакцию на конкретное возникновение исключения, для одного и того же исключения можно объявить несколько различных реакций, учитывающих соответствующий контекст.

Определяемые исключения явно вводятся программистом посредством объявления исключения. Например, объявление

```
объект_пуст, ошибка_в_данных : exception;
```

вводит два исключения (exception). Очень похоже на объявление двух объектов предопределенного типа exception. Такое объявление можно считать спецификацией исключения (хотя оно так не называется). Программист обязан задать также хотя бы одну реакцию на введенное исключение (примеры чуть ниже). Совокупность таких реакций играет роль «тела» («реализации») объявленного программистом исключения.

Таким образом, для исключений также действует принцип разделения спецификации и реализации. Ловушку (в которой размещаются реакции на исключения) также естественно считать объявлением.

Вопрос. Что естественно считать «использованием» исключения?

Определяемые исключения возникают в момент, явно указываемый в программе посредством оператора исключения (raise). Например, результатом исполнения оператора

```
raise ошибка_в_данных;
```

служит возникновение исключения ошибка_в_данных.

Факт возникновения исключения переводит исполнителя в новый режим, режим обработки исключения – происходит так называемое «распространение» исключения (ищется подходящая «ловушка исключений»), а затем выполняется «реакция на исключение», описанная в найденной ловушке. В этом режиме в основном и действуют упоминавшиеся «априорные правила поведения исполнителя». Они определяют, как найти реакцию на исключение и что делать после выполнения предписанных в ней действий.

Как уже сказано, именно на выбор этих правил влияет принцип минимальных повреждений. Рассмотрим эти правила.

8.3.2. Распространение исключений. Принцип динамической ловушки

Итак, с каждым исключением может быть связана серия ловушек, содержащих реакции на исключение и расположенных в различных программных конструктах. С учетом принципа минимальных повреждений в современных ЯП принят принцип динамического выбора ловушки – всегда выбирается реакция на возникшее исключение из ловушки, динамически ближайшей к месту «происшествия», то есть в режиме распространения исключения существенно используется динамическая структура программы.

Другими словами, конкретные действия исполнителя зависят от динамической цепочки вызовов, ведущей к тому программному конструкту, в котором возникло исключение. Поэтому в соответствующей реакции появляется возможность учесть динамический, а не только статический контекст чрезвычайной ситуации. Это, конечно, помогает предотвратить распространение повреждений.

Поясним принцип динамической ловушки на примере фрагмента программы:

```
procedure P is
   ошибка : exception;
   procedure R is
   begin
      . . . --(1)
   end R;
   procedure Q
   begin
      R;          -- вызов процедуры R;
      . . . --(2)
      exception    -- первая ловушка исключений
         . . .
         when ошибка => PUT(«ОШИБКА в Q»);
         -- реакция на исключение
         -- «ошибка» в первой ловушке
   end Q;
begin
         . . . -- (3)
      Q; — вызов процедуры Q
      . . .
exception — вторая ловушка
      . . .
      when ошибка => PUT(«ОШИБКА в P»);
      -- другая реакция на то же исключение во второй ловушке
end P;
```

Если исключение «ошибка» возникнет на месте (3), то сработает реакция на это исключение во второй ловушке и будет напечатано «ошибка в P». Если то же исключение возникнет на месте (2), то есть при вызове процедуры Q и не в R, то сработает реакция в первой ловушке и будет напечатано «ошибка в Q».

Пока подбор реакции как по динамической цепочке вызовов, так и по статической вложенности конструктов дал одинаковые результаты.

А вот когда исключение «ошибка» возникает на месте (1) в теле процедуры Q (при вызове процедуры R, в которой ловушки нет), то отличие динамического выбора от статического проявляется наглядно. Статический выбрал бы реакцию из второй ловушки в теле P, а динамический выберет реакцию из первой ловушки в теле Q.

Будет напечатано «ошибка в Q», что существенно точнее отражает суть случившегося. Именно для того, чтобы можно было точнее, конкретнее реагировать на исключение, и принят практически во всех ЯП принцип динамической ловушки.

Обратите внимание, объявляются исключения статически, подобно переменным и процедурам, а реакции на них выбираются динамически из статически определенного множества возможных реакций.

> Если разрешить вводить новые имена исключений динамически, то следовало бы создавать динамически и реакции на них, то есть динамически создавать программу. Такого рода возможности противоречат концепции статического контроля и в современных языках индустриального программирования практически не встречаются.
> Вообще, в современных ЯП поведение исполнителя в режиме обработки исключений довольно жестко регламентировано. Нет прямых аналогов, например такой житейской возможности, как сообщить «начальству» и ждать распоряжений «на месте происшествия». Или позвонить сразу и в милицию, и в скорую помощь, и в пожарную охрану, одновременно принимая собственные меры. Конечно, всякое поведение можно моделировать, но, например, несколько исключений в одном месте возникнуть не могут.

8.3.3. Реакция на исключение – принципы пластыря и катапульты

Принятая в ЯП стратегия обработки исключений прямо связана со взглядом на сущность исключений. Этот взгляд, в свою очередь, зависит от важнейших требований, определивших авторскую позицию при создании ЯП. Хотя в итоге различия поведения могут показаться не такими уж значительными, рассмотреть их обоснование и интересно, и поучительно. Выберем для определенности два ЯП: ПЛ/1 и Аду. Скажем заранее, что различия касаются лишь продолжения работы после реакции на исключение.

Принцип пластыря. Сначала о ПЛ/1. Несколько упрощая, можно сказать, что один из основных принципов конструирования языка ПЛ/1 – «предпочитать такие истолкования конструкций, которые позволяют оправдать их дальнейшее исполнение». В соответствии с этим принципом введены многочисленные правила подразумеваемых преобразований данных, допустимы различного рода сокращения и т. п.

В согласии с этим принципом исключение трактуется как относительно редкое, но в целом естественное для выполняемого конструкта событие. При его обработке следует направить усилия на скорейшее возобновление прерванного процесса. Эти усилия можно наглядно представить себе как наложение пластыря на «рану». Естественная модель поведения – прервать исполняемый процесс, вызвать «врачебную бригаду», после окончания «лечения» продолжить прерванный процесс. Обратите внимание на три составляющих поведения – прервать, вызвать (это значит, понять, кого вызвать, – врачебная бригада подбирается динамически) и продолжить.

Подобный взгляд полезен, например, при нехватке памяти – нужно вызвать подпрограмму сборки мусора или подпрограмму динамического выделения памяти, а затем попытаться продолжить работу. Описанное отношение к сущности исключения можно назвать **принципом пластыря**.

Однако где гарантии, что «заклеенный» процесс сможет нормально работать? Если исключение связано с окончанием файла или нарушением диапазона, то бессмысленно продолжать работу прерванного процесса. В ПЛ/1 в таких случаях в реакции на исключение (после «лечения», если оно требуется) применяют передачу управления туда, откуда признано разумным продолжать работу. Например:

```
ON ENDFILE GOTO Ml;
```

По многим причинам это далеко не лучшее решение. Одна из основных причин – в том, что динамическая структура программы оказывается слабо связанной с ее статической структурой. Чтобы разобраться в программе, приходится «прокручивать» каждый оператор. Короче говоря, решение с передачей управления в общем случае затрудняет чтение и отладку программ. По «неструктурированности» это решение можно сравнить с выходом из подпрограммы не по возврату, а по передаче управления. Что при этом происходит с динамической цепочкой вызовов? Остается только гадать или определять, руководствуясь «тонкими» правилами!

Итак, будем считать обоснованным, что решение с передачей управления противоречит концепции «структурного программирования». В Аде стараются обойтись без goto, тем более что таким способом «далеко не уйдешь», – в этом ЯП самая вложенная последовательность операторов, окружающая объявление метки, должна окружать и оператор перехода на эту метку. А без передачи управления принцип пластыря не позволяет адекватно обрабатывать многие виды исключений.

Упражнение. Приведите соответствующие примеры.

Принцип катапульты. Одно из ключевых требований к языку Ада – способствовать надежному программированию. Другими словами, следует стремиться к минимуму отказов из-за ошибок в программе и в данных. Когда же отказ неизбежен, то следует обеспечить по меньшей мере осмысленную диагностику.

Требование надежности оправдывает трактовку исключения как свидетельства полной непригодности «аварийного процесса» (процесса, где возникло исключение) к нормальной работе в создавшихся условиях. Стремясь к минимуму отказов, следует не «лечить» аварийный процесс, а нацелить обработку исключения на локализацию последствий «аварии», на создание возможности продолжать работу тех (связанных с аварийным) процессов, которых авария пока не коснулась.

Саму обработку исключения в «аварийном» процессе обычно разумно рассматривать скорее не как «лечение», а как «посмертную выдачу» – попытку сохранить как можно больше сведений для анализа ситуации на уровне иерархии, принимающем решения о последующих действиях.

Именно такой принцип действует в Аде (ведь надежность – одна из основных целей этого ЯП), а также в ЯП Эль-76 для машин серии Эльбрус.

При такой цели естественная стратегия – последовательно признавать аварийными вложенные процессы (начиная с самого внутреннего) до тех пор, пока среди них не найдется процесс, в котором приготовлена реакция на возникшее исключение. При этом аварийные процессы, в которых нет нужной реакции, последова-

тельно завершаются аварийным способом («катапультированием»). Найденная в конечном итоге реакция на исключение призвана обеспечить нормальное продолжение работы уцелевших процессов (и, возможно, выдачу сообщения об ошибке).

Итак, никакого возврата к аварийному процессу при такой стратегии нет, а значит, нет и опасности вызвать «лавину» исключений и сообщений об авариях. Если ведущий процесс сочтет возможным, он может снова запустить (в новых условиях) и бывший аварийный процесс. Но решение об этом не встроено в семантику ЯП, а программируется на уровне иерархии, высшем по отношению к аварийному процессу.

Назовем описанный принцип обработки исключений **принципом катапульты**. Название связано с тем, что исключение заставляет управление немедленно покинуть признанный аварийным процесс, приняв меры к спасению самой ценной информации (вполне аналогично тому, как катапультируется с аварийного самолета летчик, спасая самое ценное – человеческую жизнь).

Именно этот принцип поведения исполнителя в исключительных ситуациях воплощен в Аде (как целиком отвечающий требованиям к этому ЯП и, в частности, способствующий надежному и структурному программированию). Пример на стр. 188 показывает, как можно ликвидировать аварию и продолжить работу (после реакции на исключение управление покидает блок В).

8.3.4. Ловушка исключений

Итак, в зависимости от ЯП реакция на исключение может быть и «пластырем», и «катапультой». Осталось объяснить, как она устроена.

В общем случае тела подпрограмм, тела пакетов, тела задач, а также блоки содержат в конце обычной последовательности операторов еще часть, отделенную ключевым словом exception. Это и есть ловушка исключений. Она устроена аналогично оператору выбора, но вместо значений перечисляемого типа после ключевого слова when фигурируют имена исключений.

Например:

```
begin
   ... -- последовательность операторов
exception -- ловушка исключений
      when  плохо_обусловленная  | численная ошибка =>
         PUT("матрица плохо обусловлена");
      when others =>
         PUT("фатальная ошибка");
         raise ошибка;
end;
```

Альтернатива others, как обычно, выбирается в том случае, когда не выбраны остальные. В нашем примере при возникновении исключения плохо_обусловленная или численная_ошибка (первое – объявляемое, второе – предопределенное) печатается «плохо обусловленная матрица» и обработка исключения завершается (затем продолжает нормально работать динамически объемлющий процесс). Лю-

бые другие исключения будут пойманы альтернативой others, будет напечатано «фатальная ошибка», и возникнет новое исключение «ошибка» как результат работы оператора исключения (raise).

Это исключение будет распространяться в динамически объемлющих процессах, пока не попадет в ловушку (для предопределенных исключений ловушки предусмотрены в предопределенном пакете «система»). Если бы второй альтернативы не было, то любое исключение, отличное от двух указанных в первой альтернативе нашей ловушки, распространялось бы по динамически объемлющим процессам до «своей» ловушки.

8.4. Дополнительные особенности обработки исключений

Исключения в объявлениях. Когда исключение возникает в объявлениях некоторого конструкта, то оно немедленно распространяется на динамически объемлющие конструкты. Собственная ловушка конструкта рассчитана только на исключения, возникшие среди операторов конструкта. Это правильно, так как в ловушке могут использоваться объекты, которые из-за «недообработки» объявлений (ведь в них возникла авария) окажутся еще не существующими.

Приведенный выше пример показывает, что есть и еще одна причина – можно попасть в бесконечный цикл, если при возникновении исключения «ошибка» искать реакцию в той же ловушке. Напомним, что саму ловушку естественно считать объявлением, а именно объявлением «тел» исключений.

Исключения в асинхронных процессах. Интересно рассмотреть особенности распространения исключений на взаимодействующие задачи. У тела задачи нет динамически объемлющего процесса – соответствующий телу задачи процесс работает асинхронно, с ним могут быть динамически связаны посредством рандеву много других «равноправных» процессов-партнеров. С другой стороны, каждый асинхронный процесс запускается при обработке некоторого фиксированного объявления в определенном «родительском» конструкте.

Поэтому когда исключение возникает среди объявлений задачи, то эта задача аварийно завершается и на месте запустившего ее объявления (задачи) в родительском процессе возникает предопределенное исключение ошибка_взаимодействия. Другими словами, о возникшей чрезвычайной ситуации в задаче немедленно «узнает» запускающий ее процесс (например, для того, чтобы можно было заменить незапущенную задачу). Заменить можно потому, что запускаемая задача еще наверняка не успела вступить во взаимодействие со своими асинхронными партнерами (**почему?**) и их можно пока не предупреждать об аварии. Не успела она и запустить дочерние процессы-потомки (потому что они считаются запущенными только после нормального завершения обработки раздела объявлений в родительском процессе).

С другой стороны, если потенциальный клиент так и не запущенной (аварийной) задачи попытается обратиться к ее входу, то в этом клиенте на месте вызова

входа возникнет исключение ошибка_взаимодействия (такое исключение всегда возникает, если мастер, к которому пытаются обратиться, оказывается неработающим). Так что исключение в случае аварии в одном из асинхронных процессов распространяется не только «линейно» вверх по цепочке от потомков родителям, но и «веером» ко всем клиентам.

Когда же исключение возникает не в объявлениях, а в теле задачи и в своем распространении доходит до самой внешней в этом теле ловушки, то задача завершается аварийно (независимо от того, «поймано» ли исключение), но в запустившем ее процессе-родителе исключение не возникает. По-видимому, потому, что в общем случае ничего разумного он сделать не сможет – перезапускать аварийную задачу считается опасным: она уже успела поработать с партнерами, причем не только с клиентами, но и мастерами, а также запустить своих потомков.

Упражнение. Предложите версии ответа на вопрос, почему не возникает исключение в мастерах, с которыми могла взаимодействовать аварийная задача, или в ее потомках.

Подсказка. Возможно, авария не имеет к ним никакого отношения.

Заметим, что исключения, возникающие при рандеву, считаются возникшими в обоих партнерах и распространяются в них независимо и асинхронно.

Уточнение к принципу динамической ловушки. Поиск ловушки происходит с учетом динамической цепочки вызовов любых блоков, не обязательно процедур, пока она есть. Выше – с учетом статической вложенности объявлений, затем, возможно, снова динамической цепочки вызовов и т. д.

Универсальная ловушка. Описанные ловушки требуют знать и учитывать все имена потенциальных исключений – это не всегда удобно и даже возможно. Иногда нужно единообразно реагировать на любое исключение, то есть иметь средство абстракции от характера исключения, средство реагировать на сам факт его возникновения (на сам факт аварии – не важно, какой именно). Обычно это, по сути, потребность принять лишь меры общего характера, но отложить принятие конкретных решений до достижения «достаточно компетентных» уровней иерархии.

Такая потребность возникает, во-первых, на уровнях программной иерархии, которые просто некомпетентны содержательно реагировать на содержательные аварии, и, во-вторых, в таких структурах, где возникшее исключение статически невидимо (и потому они формально не могут содержать ловушку с именем такого исключения).

Упражнение. Придумайте пример такой структуры.

Пример типичной программной иерархии:

```
package обслуживание is
   нет_исполнителей, нет_ресурсов, нет_заказов : exception;      -- содержательные
                                                                  --  исключения
   procedure распределить_работу is
      ...
```

```
   raise нет_исполнителей;   -- содержательное исключение, но что делать при его
      ...                    -- возникновении, здесь не ясно.
    end распределить_работу; -- Поэтому ловушки нет.
                             -- Никакой реакции!

   procedure проверить_исполнение is
      ...
   raise нет_заказов;        -- по той же причине ловушки нет
      ...
   end проверить_исполнение;

   procedure выполнить is
      ...
   raise нет_ресурсов;       -- по той же причине ловушки нет
      ...
   end выполнить;

   procedure обслужить_категорию_A(f : файл) is
      ...                    -- «некомпетентный» уровень
   begin
      открыть(f);
      распределить_работу;
      ...
      выполнить;
      ...
      проверить_исполнение;
      ...
      закрыть(f);

   exception -- универсальная ловушка (для любых исключений)
      when others => закрыть (f); raise;
                       -- что бы ни произошло, нужно
                       -- закрыть файл, а с остальным
                       -- пусть разбираются выше
   end обслужить_ категорию_A;
      ...
exception              -- это "компетентный" уровень
   when нет_исполнителей =>
      PUT('Вас много, а я одна!);
   when нет_заказов =>
      PUT('Нет заказов, нужна реклама!);
end обслуживание;
```

Ловушка с альтернативой «when others» – это и есть ловушка, пригодная для любых исключений. Оператор raise перевозбуждает последнее возбужденное исключение, которое продолжает распространяться обычным методом. Все происходит почти так же, как если бы ловушки вообще не было. Но это «почти» – предварительная реакция соответствующих уровней программной иерархии.

Выход за границы видимости. Исключения нет_ресурсов (и др.) можно объявить и на нижних уровнях иерархии, но обрабатывать (без учета их особенностей) на верхних уровнях в универсальных ловушках. Здесь в Аде нет статического контроля. По-видимому, это считается более надежным, чем провоцировать отсутствие исключений из-за опасений, что не удастся придумать адекватную реакцию (а при отсутствии таковой пришлось бы вставлять чисто формально реакцию raise;). В других ЯП (с параметрами-процедурами) подобные случаи были бы вообще статически неконтролируемыми (**почему?**). Например:

```
package P is
   procedure f;
   procedure Pr(f : proc) is    -- параметр-процедура
      f;                        -- здесь e не видно, но может распространяться
   end Pr;      -- нет оснований контролировать наличие ловушки
                -- исключения e — ведь возможны различные параметры-процедуры
end P;
package body P is
   e : exception;
   procedure f is
      raise e;
   end f;       -- ловушки нет
end P;          -- ловушки также нет
with P; use P;
   begin
      Pr(f);    -- здесь e видно
   end;
exception
   when t => S; -- ловушки для e не оказалось
end;
```

Можно и так запрограммировать Pr:

```
procedure Pr(f; proc) is
   begin
      f;
exception          -- универсальная ловушка;
                   -- e невидимо, но обрабатывается
      when others =-> что-то; raise;
                   -- и распространяется дальше
end Pr;
```

Различия между исключениями в объявлениях и операторах. Суть различий состоит в том, что ловушка блока не обслуживает исключений, возникших при обработке объявлений этого блока. Ловушка только для операторов.

Например:

```
<< B >>
declare
      a : integer := f(22); — при вычислении  f  возможны исключения
      x : real;
begin
```

```
      ...
   exception
      when числ_ош => PUT('ОШ в блоке B');
         PUT(a); PUT(x);
end B;
→
```

Исключение, возникшее при вычислении f, немедленно распространяется в точку, указанную стрелкой. **(Почему так?)** Ведь если исключение возникло среди объявлений, то нет гарантий, что нормально введены объекты, используемые в ловушке, а задача ловушки – сохранить объекты (для анализа) присваиванием глобальным переменным или выводом. Если в такой ситуации не обойти ловушку, то ее исполнение может нанести глобальным объектам дополнительный ущерб (а также привести к новым исключениям).

Напомним, что при этом сама ловушка тоже считается объявлением, а именно «телом исключения».

Можно считать, что здесь проявляется еще один полезный принцип – **принцип минимизации каскадов** (взаимозависимых) исключений. С другой стороны, его можно непосредственно вывести из принципа минимальных повреждений.

Упражнение. Объясните смысл и обоснуйте принцип минимальных каскадов.

Подавление исключений (оптимизирующие указания). Для полноты представления об обработке исключений осталось добавить, что в случае, когда некоторые проверки ограничений, потенциально приводящие к возникновению исключений, считаются дорогими (неприемлемо ресурсоемкими), их можно отменить с помощью так называемых прагм (оптимизирующих указаний) вида

```
pragma подавить(проверка_индексов, на => таблица);
```

Если реализация способна реагировать на такие указания (они не обязательны для исполнения), то в нашем случае проверка индексов будет отменена для всех объектов типа «таблица». Можно управлять исключениями и с точностью до отдельных объектов. Конечно, подобными указаниями следует пользоваться очень осторожно. В Аде есть и другие оптимизирующие указания. Все они не обязательны.

Выводы. Итак, на примере аппарата исключений в Аде показаны в действии все три основных принципа стр. 185. В частности, этот аппарат позволяет отделить программирование содержательной функции программного сегмента от программирования его взаимодействия с другими сегментами в чрезвычайных (аварийных) обстоятельствах.

Программируя содержательную функцию, можно абстрагироваться от необычных ситуаций, а программируя взаимодействие в необычных ситуациях, в значительной степени абстрагироваться от содержательной функции сегмента, опираясь на априорные правила поведения Ада-исполнителя. Для особо важных чрезвычайных ситуаций можно заранее заготовить названия и ловушки в библиотечных модулях, а также программировать ловушки в пользовательских модулях, конкретизируя необходимую реакцию. Таким образом, концепция исключения – одна из компонент общего аппарата абстракции-конкретизации в ЯП. Ее можно

было бы довести и до уровня модульности, физически отделив ловушки от тел сегментов. По-видимому, такая возможность появится в ЯП будущего. (Точнее, уже появилась в языке SDL/PLUS [15] и в последних версиях языка Модула-2).

Стоит заметить, что исключения – весьма специфический аспект ЯП, очевидным образом нуждающийся в развитии и определении более четкого места в структуре ЯП в целом. Ведь исключения – это не сообщения, но похожи; не прерывания, но похожи; не goto, но похожи; не процедуры, но похожи; не типы, но похожи.

Действительно, это не обычные сообщения, потому что нет явного отправителя и адресата, нет явной структуры сообщения, но вместе с тем функция исключения – сообщить одному или нескольким процессам (содержащим подходящие ловушки) о чрезвычайных обстоятельствах. Это не обычные прерывания, потому что не предполагают обязательного участия «средств низкого уровня» – аппаратуры или ОС для своей обработки. Хотя исключения было бы правильно назвать «прерываниями на уровне исходного ЯП». Это не обычные процедуры, хотя исключениям соответствуют последовательности действий, вызываемых по именам (но тела связываются с именами не статически, а динамически, недопустимы параметры, необязателен вызов и т. д.). Это не обычный тип, хотя можно объявлять соответствующие «объекты», потому что нельзя передавать их каким-либо определяемым пользователем операциям (нельзя передавать как параметры).

И вся эта увлекательная специфика исключений объясняется их особой ролью – обслуживанием потребности в разделении нормального и «чрезвычайного» режимов работы программы с учетом принципов минимальных возмущений и минимальных повреждений.

Упражнение. Дополните анализ аппарата исключений в Аде с точки зрения связей с другими языковыми конструктами. Проделайте то же для других известных вам ЯП (например, Эль-76 или ПЛ/1).

Глава 9

Библиотека

9.1. Структура библиотеки 202
9.2. Компилируемый (трансляционный) модуль 202
9.3. Порядок компиляции и перекомпиляции (создания и модификации программной библиотеки) 203
9.4. Резюме: логическая и физическая структуры программы 204

9.1. Структура библиотеки

До сих пор мы избегали подробного описания языковых свойств, ограничиваясь сведениями, достаточными для демонстрации рассматриваемых концепций и принципов. Однако в будущем мы намерены существенно затронуть авторскую позицию, для которой, конечно, важны все тонкости ЯП (иначе они бы в нем не появились). Более того, мы намерены изложить принципы, в определенном смысле управляющие сложностью создаваемого ЯП. Для их понимания необходимо, чтобы читатель был в состоянии в деталях сопоставить решения, принятые авторами различных ЯП.

Поэтому в ближайших разделах, завершая знакомство с основными языковыми абстракциями, мы подробно остановимся на избранных аспектах Ады, а именно на раздельной компиляции, управлении видимостью идентификаторов и обмене с внешней средой. Основная цель упоминания подробностей – продемонстрировать сложность языка и возникающие в этой связи проблемы. Заинтересованного читателя отсылаем к руководствам по Аде [16, 17, 18].

Ранее мы рассмотрели виды связывания раздельно транслируемых модулей в Аде. Посмотрим на ту же самую проблему немного с другой стороны – обсудим устройство программной (трансляционной) библиотеки. Ада – первый ЯП, в котором особенности использования библиотеки тщательно проработаны и зафиксированы в определении ЯП. В этом отношении полезно сравнить Аду, например, с Фортраном.

9.2. Компилируемый (трансляционный) модуль

Компилятор получает «на вход» компилируемый модуль, который состоит из (возможно, пустой) спецификации контекста и собственно текста модуля. Спецификация контекста содержит указатели контекста (with) и сокращений (use). Займемся первым.

Уже было сказано, что with – средство явного указания односторонней связи: в использующем модуле перечисляют имена необходимых ему библиотечных модулей.

Таким образом, любое имя, используемое в данном модуле, должно быть либо объявлено в самом этом модуле или в связанных с ним (при помощи двусторонней связи!) библиотечных или родительских модулях; либо объявлено в пакете STANDARD; либо предопределено; либо явно перечислено в указателе контекста (with).

Итак, «пространство имен» модуля ограничено и явно описано.

Упражнение. Сравните с EXTERNAL в Фортране. В чем отличия?

В указателе контекста необходимо перечислять лишь непосредственно используемые имена библиотечных модулей, то есть те имена, которые явным обра-

зом присутствуют в тексте модуля. Например, если библиотечная процедура P использует библиотечную процедуру Q, а та – в свою очередь, библиотечный пакет R, то соответствующие компилируемые модули должны иметь следующий вид:

```
with R;                    with Q;
procedure Q is             -- with R писать не надо!
   ...                     procedure P is
begin                      begin
   ...                        ...
   R.P1;                      Q; -- вызов Q;
   -- вызов процедуры,        ...
   -- описанной в R;       end P;
   ...
end Q;
```

Правила ЯП не запрещают указывать имя «лишнего» модуля, однако, как мы далее увидим, это не просто бессмысленно, но и опасно.

9.3. Порядок компиляции и перекомпиляции (создания и модификации программной библиотеки)

Очевидно, что этот порядок не может быть абсолютно произвольным. Нам уже известно все, чтобы сформулировать требования к нему. Выделим две группы требований, обусловленные двусторонним и односторонним связываниями соответственно. Напомним, что в исходном состоянии библиотека содержит лишь предопределенные библиотечные модули.

Двусторонние связи:

1. Тело следует компилировать после спецификации.
   Следствия. После перекомпиляции спецификации необходимо перекомпилировать тело. Перекомпиляция тела не требует перекомпиляции спецификации.
2. Вторичный модуль следует компилировать позже соответствующего родительского модуля.
   Следствие. Перекомпиляция родительского модуля влечет перекомпиляцию всех его вторичных модулей.

Односторонние связи:

Использующий модуль следует компилировать позже используемого (то есть модуль можно компилировать только после компиляции (спецификаций, не тел!) модулей, перечисленных в его указателе контекста.

Следствия. После перекомпиляции библиотечного модуля (спецификации, а не тела) необходимо перекомпилировать все использующие модули. Перечислять «лишние» модули в указателе контекста действительно вредно!

Вопрос. А как в Фортране (где компиляция модулей независимая)?

Реализации дано право квалифицированно «разбираться в ситуации» и выявлять (для оптимизации) те перекомпиляции, которые фактически не обязательны.

Вопрос. За счет чего?

9.4. Резюме: логическая и физическая структуры программы

В Аде следует различать физическую и логическую структуры программы. Эти понятия тесно связаны, но не эквивалентны. Логическая структура – это абстракция физической структуры, а именно абстракция от конкретного способа разбиения на (физические) модули.

Во всех случаях, когда важен смысл, а не особенности жизненного цикла программы (создание, компиляция, хранение, модификация), нас интересует логическая структура программы. Однако в остальных случаях приходится учитывать (и использовать) ее физическую структуру.

Физическая структура программы образуется совокупностью (компилируемых) модулей, отделенных друг от друга и «подаваемых» компилятору в определенном порядке. Логическую структуру готовой (завершенной) программы образуют сегменты. (Иногда их называют «программные модули».) Здесь уже совершенно не важен порядок компиляции. Более того, при переходе от физической к логической структуре меняется само понятие компоненты программы. Так, с точки зрения физической структуры, спецификация и тело (библиотечного) пакета – это разные модули, а в логической структуре это единый сегмент – пакет, в котором при необходимости выделяется, например, видимая часть, не совпадающая в общем случае со спецификацией. **(Из-за чего?)**

На уровне логической структуры фактически пропадает разница между первичными и вторичными модулями, а также становится ненужным само понятие библиотеки.

Каковы же средства создания логической структуры программы над ее физической структурой? Часть из них мы рассмотрели – это способы одностороннего и двустороннего связывания: правила соответствия между спецификацией и телом библиотечного модуля, заглушки и (полные) имена родительских модулей в заголовках вторичных модулей, указатель контекста.

Другую группу средств связывания образуют правила видимости идентификаторов и правила идентификации имен. Об этом – в следующем разделе.

Глава 10

Именование и видимость (на примере Ады)

10.1. Имя как специфический знак 206
10.2. Имя и идентификатор 206
10.3. Проблема видимости 206
10.4. Аспекты именования 207
10.5. Основная потребность и определяющие требования 207
10.6. Конструкты и требования, связанные с именованием 208
10.7. Схема идентификации 210
10.8. Недостатки именования в Аде 216

10.1. Имя как специфический знак

Идентификация и видимость имен – это два важных аспекта общей проблемы именования в ЯП, которую мы и рассмотрим на примере Ады, выделяя по возможности общие принципы и концепции.

Начнем с основных терминов: имя и идентификация имени. Программа на ЯП представляет собой иерархию знаков. Для некоторых знаков если не сам денотат, то хотя бы его класс определяется в значительной степени по структуре знака (оператор присваивания, цикл, объявление процедуры).

Имя как специфический знак характеризуется тем, что по одной его структуре (то есть только по внешнему виду знака, без привлечения контекста) в общем случае нельзя получить никакой информации о денотате.

Например, по одному только знаку "A", "A.B.C.D" или "A(B(C(D)))" в Аде невозможно сказать не только то, что конкретно он обозначает, но даже приблизительно определить класс обозначаемой сущности (процедура, переменная, тип, пакет и т. п.).

Итак, во-первых, имя бессмысленно без контекста. Во-вторых, оно служит основным средством связывания различных контекстов (фрагментов программы) в единое целое. В-третьих, денотат имени можно извлечь только из анализа контекста. Процесс (правила, алгоритм, результат) такого анализа называют **идентификацией имени**.

10.2. Имя и идентификатор

В Аде любое имя содержит хотя бы один идентификатор. Идентификатор – это атомарное имя (то есть никакое имя не может быть частью идентификатора). Идентификатор в Аде строится, как и в других ЯП, из букв и цифр, которые (в отличие от многих других ЯП) можно разделять единичными подчеркиваниями.

Только идентификаторы можно непосредственно объявлять в программе. Денотаты других имен вычисляются через денотаты их компонент-идентификаторов.

> В Аде особую роль играют предопределенные знаки операций (+ , – и др.). Их можно использовать только в строго определенных синтаксических позициях, а именно в позициях операций в выражениях. Соответственно, и переопределять такие знаки разрешается только с учетом указанного требования. Все это делается, чтобы обеспечить стабильность синтаксиса (привычка программиста – гарантия надежности, да и анализ выражения проще). Для знаков операций проблема идентификации стоит так же, как и для обычных идентификаторов. Мы не будем их особо отличать.

10.3. Проблема видимости

Вхождение идентификатора в Ада-программу может быть либо определяющим, либо использующим. Во всяком ЯП с достаточно сложной структурой программы существует проблема установления соответствия между определяющими и ис-

пользующими вхождениями. Следуя адовской терминологии, будем называть ее **проблемой видимости идентификаторов**. Полная проблема идентификации имен включает проблему видимости идентификаторов, но не сводится к ней.

Вопрос. В чем различие?

Подсказка. Имена бывают не только идентификаторами. К тому же мало найти определяющее вхождение, нужно еще вычислить денотат.

Заметим, что следует различать статическую и динамическую идентификации. Так, если объявлено

```
A: array(1..10) of INTEGER;
I: INTEGER;
```

то со статической точки зрения имя A(I) обозначает элемент массива A, но динамическая идентификация при I=3 даст A(3) (то есть 3-й элемент), а при I=11 – исключение нарушение_диапазона.

10.4. Аспекты именования

Именование – средство построения логической структуры программы над ее физической структурой в том смысле, что после того, как в компилируемом модуле идентифицированы все имена, он становится составной частью теперь уже логической структуры программы, поскольку оказывается связанным со всеми используемыми в нем понятиями и элементами программного комплекса как единого целого.

Выделим следующие относительно независимые аспекты именования: разновидности объявлений; строение имен; строение «пространства имен»; правила видимости идентификаторов; схема идентификации имен. Всюду ниже в этом разделе будем игнорировать физическую структуру программы (ее разбиение на модули). Учитывать будем лишь ее логическую структуру (разбиение на сегменты). Таким образом, исключаем из рассмотрения имена модулей и их связывание.

Применим к проблеме именования уже не раз нами использованный принцип технологичности: **от технологической потребности через определяющие требования к выразительным средствам (языковым конструктам)**.

10.5. Основная потребность и определяющие требования

Основная «внешняя» технологическая потребность очевидна – точно называть необходимые компоненты программы. Однако поскольку эти компоненты разнородны и обслуживают весьма разнообразные потребности, то существует сложное и многогранное «внутреннее» определяющее требование: именование должно быть хорошо согласовано со всеми средствами ЯП и должно отвечать общим требованиям к нему (надежность, читаемость, эффективность и т. п.).

Другими словами, концепция именования и основные конструкты ЯП (а также заложенные в них концепции) взаимозависимы.

> Сложные и многообразные конструкты ведут к сложному именованию, и наоборот, относительно простые способы именования требуют относительной простоты конструктов (сравните именование в Аде с именованием в Бейсике или Фортране). Искусство автора ЯП проявляется в умении найти разумный компромисс между собственной сложностью ЯП и сложностью его использования для сложных задач (Фортран или Бейсик относительно просты, но сложные задачи на них программировать труднее, чем на Аде).

При создании Ады приоритет имела задача включения в него богатого набора средств (конструктов), позволяющих адекватно реализовывать большой набор технологических потребностей, причем для многих технологических потребностей уже в самом ЯП заготавливалось специальное средство (например, потребность в родовой настройке может быть удовлетворена специализированным макропроцессором, а не встраиваться в ЯП непосредственно).

В результате именование получилось довольно сложным. Это признают и авторы языка (Ледгар совместно с Зингером даже отстаивали идею стандартного подмножества Ады, чего никак не хотел допустить заказчик – МО США [32]). Значительная часть критических замечаний в адрес Ады также касается идентификации имен.

10.6. Конструкты и требования, связанные с именованием

Перечислим общие требования к языку и те конструкты Ады, которые оказались существенно связанными с именованием.

Требования. Глубокая **структуризация** языковых объектов, **раздельная** компиляция, относительная **независимость именования** внутри сегментов, необходимость **переименования** и **сокращения** длинных имен.

Кроме того, **критичность проблемы полиморфизма** потребовала заменить классический (бытовавший в ЯП еще со времен Алгола-60) запрет объявлять одинаковые идентификаторы в одном блоке более гибким ограничением, позволяющим объявлять одноименные операции, процедуры и функции с различающимися профилями.

Принцип **обязательности объявлений** для всех имен (кроме предопределенных) в сочетании с необходимостью **производных типов** привел к так называемым неявным объявлениям операций. Например:

```
package P is
   type T is(A,B);
   procedure Q(X : in T, Y : out INTEGER);
end P;
      ...
      type NEW_T is new T;
      ...
```

Тип NEW_T должен обладать свойствами, аналогичными всем свойствам типа T. В частности, должен иметь два перечисляемых литерала A и B (теперь уже типа NEW_T) и операцию-процедуру P с параметрами

```
(X : in NEW_T, Y : out INTEGER).
```

Чтобы не заставлять программистов переписывать соответствующие объявления **(чем это плохо?)** и вместе с тем соблюсти принцип обязательности объявлений, авторы Ады были вынуждены ввести так называемые **неявные объявления**. Указанные выше литералы и процедура считаются объявленными неявно.

> **Вопрос.** А зачем обязательность объявлений?
>
> **Подсказка.** Для прогнозирования-контроля и, следовательно (по замыслу), повышения надежности.

Наконец, иногда оборачивается неприятными сюрпризами **требование определенного «комфорта» при написании программ**. В результате возникает много локальных неоднозначностей. Скажем, «A(I)» может обозначать элемент массива, вызов функции или вырезку массива ((n–1)-мерный подмассив n-мерного массива A).

Следующий пример взят из журнала Ada LETTERS. Неприятность связана с неявными инициализирующими выражениями у входных параметров функции и процедур.

```
procedure test is
      type Enum is(Red, Green);
      type Vec is array(Enum) of Enum;
      X : Enum;
      Y : Vec;
      function F(A : Enum := Red) return Vec is
      begin
         return Y;
      end;
begin
   X := F(Red);
   -- Что в последней строчке? Вызов функции с параметром RED или элемент массива,
   -- вычисленного вызовом функции без параметров (ведь инициализированные
   -- параметры можно опускать).
   -- [Надо бы F()(Red), как в Фортране-77].
   Y := F(Red); -- здесь тоже не ясно
                -- следует учесть, что правилами перекрытия пользоваться некорректно —
                -- функция одна, и перекрытия нет
end test;
```

> **Замечание.** Конечно, так программировать нельзя независимо от свойств ЯП. Программа не ребус. Ее нужно читать, а не разгадывать!

Еще хуже:

```
procedure F is
   type ARR;
```

```
   type ACC is access ARR;
   type ARR is array(1..10) of ACC;
   X : ACC;
   function f(X : INTEGER := 0) return ACC is
   begin
      return new ARR;
   end f;
begin
   X := f(1); — допустимы две интерпретации
end;
```

Вопрос. Какие именно интерпретации?

Итак, требования к языку, которые в наибольшей степени повлияли на схему идентификации в Аде, названы. Рассмотрим эту схему.

10.7. Схема идентификации

10.7.1. Виды объявлений в Аде

Изложенные ниже подробности имеют основной целью продемонстрировать относительную сложность идентификации в Аде, а не полностью описать ее или тем более научить ею пользоваться. Поэтому читатель, для которого доказываемый тезис очевиден или неинтересен, может без ущерба пропустить оставшуюся часть главы 10.

Явные объявления. Кроме собственно объявлений, будем считать явными объявлениями также части объявлений, синтаксически не выделяемых в отдельные конструкты-объявления, хотя содержательно играющие такую роль. Это компоненты записей (в том числе и дискриминанты типа), входы задач, параметры процедур и родовые параметры, перечисляемые литералы, параметр цикла.

Неявные объявления. Это имя блока, имя цикла, метка оператора, перечисляемые литералы и унаследованные подпрограммы производных типов, предопределенные операции типов различных категорий. Перечисляемые литералы считаются неявно объявленными функциями без параметров.

Зачем нужны неявные объявления. Как уже не раз отмечалось, одним из важнейших требований к Аде было требование надежности, составной частью которого является требование обнаруживать и диагностировать как можно больше нарушений во время компиляции (до начала выполнения программы), то есть требование **статического прогнозирования и статического контроля**. Это включает и контроль использования имен (идентификаторов), для чего и необходимо прогнозирование, то есть тот или иной способ объявления.

Явно объявлять метки (как в Паскале) обременительно. С другой стороны, метки могут конфликтовать с другими именами; чтобы контролировать такие коллизии с учетом областей локализации, удобно считать метки объявленными «рядом» с остальными (явно объявленными) именами рассматриваемой области локализации.

Например, метки операторов считаются неявно предобъявленными непосредственно после всех явных объявлений соответствующей области локализации и тем самым действуют в пределах всей этой области.

Неявные объявления унаследованных подпрограмм. Основные неприятности возникают из-за неявных объявлений унаследованных подпрограмм при объявлении производных типов. По замыслу производного типа в момент своего объявления он должен сохранить все те свойства (кроме имени), которые заданы в определяющем пакете родительского типа. Следовательно, вместе с новым типом должны считаться объявленными и подпрограммы, унаследованные от родительского типа. Но объявленными где? Ведь явных объявлений этих подпрограмм для нового типа нет, а от точного места объявлений зависит и смысл программы, и результаты контроля.

Упражнение. Приведите примеры такой зависимости.

В Аде эта проблема решается так: все унаследованные подпрограммы считаются неявно объявленными сразу вслед за объявлением производного типа. Эти неявные объявления «уравнены в правах» с явными объявлениями.

Далее, если родительский тип был перечисляемым, то производный должен иметь те же перечисляемые литералы; поэтому их тоже считают объявленными неявно (в качестве функций без параметров).

Вопрос. При чем здесь функции, да еще без параметров?

Подсказка. Эти литералы могут перекрываться, поэтому их удобно считать функциями для единообразия правил перекрытия.

Вопрос. А почему можно считать литералы функциями?

Другие особенности механизма объявлений

1. Механизм наследования во взаимодействии с указателем сокращений и перекрытием оказывается довольно сложным. Подробнее об этом будет сказано в связи с проблемой сокращений. Основная идея: наследуются только подпрограммы из определяющего пакета родительского типа, причем (важно!) наследование возможно только извне пакета.
2. Формальная область действия неявного объявления метки может оказаться существенно шире, чем та часть текста, из которой на эту метку можно ссылаться. Например, запрещено передавать управление внутрь цикла, хотя формально внутренняя метка цикла может считаться неявно объявленной вне цикла, после явных объявлений внешней области локализации.
3. Наравне с идентификаторами объявляются строки – знаки операций и символьные константы.

Устройство полных (составных) имен. Общая структура имени такова:

```
<нечто-без-точки>{.<нечто-без-точки>}
```

где <нечто-без-точки> – это <идентификатор>{нечто-в-скобках}

В скобках могут быть записаны либо индексы массива, либо аргументы вызова функции. Заметим, что в Аде возможен элемент массива вида a(i)(j)...

Рассмотрим три примера:

```
procedure P is
   type T  is(A,B,C);
   type T1 is array(1..10) of T;
   type T2 is record
      A2 : T1;
      B2 : T1;
   end record;
   type T3 is array(1..10) of T2; — Массив записей сложной структуры
   X : T3;
begin
   X(2).A2(3):=C;
end;
procedure Q is
   package P1 is
      package P2 is
         type T is
         ...
      end P2;
   end P1;
   X : P1.P2.T; — Способ «достать» из пакета нужный тип
...
end Q;
procedure P is
   I : INTEGER;
   ...
   procedure P1 is
      I : INTEGER;
      ...
   begin
   ...
      P.I:=P1.I;
      P1.I:= P.P1.I+1;        (*)
      --  эквивалентно    I:=I+1;
   end P1;
end P;
```

Последний пример одновременно демонстрирует и мощь средств именования в Аде. В традиционных ЯП с блочной структурой имя, объявленное во внутренней области локализации, закрывает все глобальные омонимы. Другими словами, одноименные объекты в такой области локализации абсолютно недоступны.

В общем случае это не всегда удобно; иногда полезно иметь доступ к закрытому объекту (приведите примеры, когда это может понадобиться!).

В Аде достаточно указать полное имя закрытого объекта. Но для этого необходимо иметь возможность называть именами области локализации. Поэтому в Аде появились именованные блоки и циклы. (В цикле с параметром объявляется параметр цикла; имя цикла применяется в операторе выхода из цикла (exit).)

Упражнение. Приведите примеры применения полных имен для доступа к закрытым локальным переменным блока, к закрытым параметрам цикла, для выхода из вложенного цикла. Воспользуйтесь при необходимости каким-либо учебником по программированию на Аде.

Вопрос. Как вы думаете, разрешен ли в Аде доступ по полным именам извне внутрь области локализации? Постарайтесь ответить, не пользуясь руководством по Аде, опираясь только на свое понимание общих принципов этого ЯП.

Итак, денотат полного имени получается последовательным уточнением при движении по составному имени слева направо.

Применение составных имен. Составное имя может использоваться в следующих случаях:

1) именуемая компонента – это компонента объекта комбинированного типа, или вход задачи, или объект, обозначаемый ссылочным значением;
2) полное имя – это объект, объявленный в видимой части пакета, или объект, объявленный в охватывающей области локализации.

Источники сложности. В именуемой компоненте указанные случаи с точки зрения контекстно-свободного синтаксиса не различаются. К тому же в одном и том же полном имени могут комбинироваться несколько случаев.

Пример:

```
procedure P is
   package Q is
      type T is record
         A : INTEGER;
         B : BOOLEAN;
      end record;
      X:T;
   end Q;
begin
   ...
   Q.X.A:=1;
end P;
```

10.7.2. Области локализации и «пространство имен» Ада-программы

Как известно, областью локализации называется фрагмент программы, в котором введены имена, непосредственно доступные в этом фрагменте и непосредственно недоступные вне этого фрагмента. В Аде имеются следующие разновидности областей локализации:

- программный модуль (спецификация плюс тело);
- объявление входа вместе с соответствующими операторами приема входа (вводятся имена формальных параметров);
- объявление комбинированного типа (вводятся имена полей) вместе с соответствующим возможным неполным объявлением или объявлением при-

ватного типа (вводятся дискриминанты), а также спецификацией представления;
- переименование (возможно, вводятся новые имена формальных параметров для новых имен подпрограмм);
- блок и цикл.

Область локализации физически не обязана быть связным фрагментом. В частности, возможны области локализации, состоящие из нескольких компилируемых модулей.

Омографы и правила перекрытия. Отличительные особенности областей локализации в Аде – применение полных имен для доступа к непосредственно невидимым объектам и применение перекрытия для борьбы с коллизией имен. Примеры первого уже были. Займемся подробнее вторым.

В Аде разрешено перекрывать имена операций и процедур (включая предопределенные операции, такие как +, –, *, and), перечисляемые литералы (поэтому они и считаются неявно объявленными функциями без параметров) и входы задач (вспомните, вызов входа по форме ничем не отличается от вызова процедуры). Для краткости будем в этом разделе называть имена всех указанных категорий перекрываемыми именами.

Профилем перекрываемого имени (и соответствующего объявления) называется кортеж, составленный в естественном порядке из формальных параметров и результата соответствующей подпрограммы (указываются имена и типы параметров и результата, если он имеется).

Два объявления одинаковых имен называются **омографами**, когда их профили совпадают либо когда одно из имен не относится к классу перекрываемых.

Основное ограничение: омографы в одной области локализации в Аде запрещены. Вместе с тем во вложенной области локализации имя может быть закрыто только омографом, а не просто объявлением такого же имени.

Пример:

```
procedure P is
   function F(X : FLOAT) return INTEGER;
   I: INTEGER;
   ...
   procedure P1 is
      function F(X : INTEGER ) return INTEGER;
      ...
      begin
         I := F(1.0); — эквивалентно I:=P.F(1.0)
         ...
         I := F(1); — эквивалентно I:=P1.F(1)
      end F;
      ...
   end P1;
   ...
end P;
```

10.7.3. Область непосредственной видимости

Какие же объявления непосредственно видимы из данной точки программы? Начнем с проблемы видимости для простых имен: определить область (непосредственной) видимости – множество имен, (непосредственно) видимых (доступных без уточнений с помощью разделителя «.») из некоторой точки программы.

Область видимости в некоторой точке компилируемого модуля конструируется по следующей схеме из последовательно объемлющих друг друга частей (буквы на схеме соответствуют перечисленным ниже частям области видимости):

```
{-д-{-г-{-в-{-б-{-а-}-б-}-в-}-г-}-д-},
```

где

а – объявления, текстуально входящие в рассматриваемый модуль и видимые из рассматриваемой точки по традиционным правилам блочной структуры (с учетом Ада-обобщения этого понятия на области локализации – вспомните циклы, объявления комбинированных типов и т. п.);

б – объявления из предопределенного пакета STANDARD, не закрытые объявлениями из части (а);

в – контекст модуля, задаваемый двусторонними связями с программной библиотекой; другими словами, рассматриваются объявления родительских модулей, если рассматриваемый модуль – вторичный, и в область видимости добавляются только те из них, которые не закрываются частями (а) и (б);

г – имена библиотечных модулей, упомянутых в указателе контекста рассматриваемого модуля;

д – объявления из видимой части тех пакетов, имена которых перечислены в видимых указателях сокращений и оказались не закрытыми на предыдущих этапах. Пример:

```
package P is -- первичный библиотечный пакет
   I : INTEGER;
end P;
with P; use P;
procedure Q is
   -- какова область видимости здесь?
   package R is
      B : BOOLEAN;
   end R;  -- а здесь?
   use R;
   begin
      B := TRUE;
      I := 1;
end Q;
```

Вопрос. Зачем на схеме указаны две компоненты каждой из частей (б)...(д)? Каким объявлениям они могут соответствовать?

Подсказка. Не все объявления предшествуют использованию.

10.7.4. Идентификация простого имени

Итак, область видимости построена. Теперь нужно выбрать подходящее объявление. Если имя перекрыто, получаем несколько вариантов объявления. Чтобы отобрать нужное, необходимо исследовать контекст использующего вхождения имени. Тут много дополнительных правил, на которых останавливаться не будем. Если в конечном итоге контекст позволяет отобрать ровно одно объявление, то перекрытие считается разрешенным, а имя – полностью идентифицированным; в противном случае имеет место нарушение языковых законов.

10.7.5. Идентификация составного имени

Идентификация составного имени сводится к последовательной идентификации простых имен и вычислению соответствующих компонент структурных объектов при движении по составному имени слева направо.

Пример:

```
with PACK; use PACK;
procedure P is
      package Q is
         type T is record
            A : T1;
            B : T2;  -- T1 и T2 объявлены в PACK
         end record; end Q; use Q; X : T; Y : T1;
   procedure PP is
      X : FLOAT;
      ...
   begin
      P.X.A.:= Y;  -- все правильно (почему?)
   end PP;
end P;
```

Заметим, что перекрытие может «тянуться» (и «множиться»!) при анализе вдоль имени. Например, перекрытое имя функции может соответствовать двум (или нескольким) потенциальным результатам, каждый из которых может иметь сложную структуру. Это заставляет анализировать каждый из них. Но при этом анализе для продвижения по полученным структурам может снова потребоваться учитывать потенциальные результаты перекрытых функций и т. д. Конечно, так программировать нельзя (ненадежно!), но язык позволяет.

Упражнение. Придумайте соответствующие примеры перекрытий.

10.8. Недостатки именования в Аде

Основные принципы именования в языке Ада рассмотрены. Очевидно, что идентификация достаточно сложно устроенного имени оказывается нетривиальной. Но не это самое страшное. Оказывается, что именование в Аде не удовлетворяет

одному из основных требований к этому языку – надежности. Основных источников ненадежности два. Во-первых, пространство имен компилируемого модуля может формироваться неявно (вспомните об операциях производных типов). Во-вторых, смысл указателя сокращений определен столь неестественно, что оказался нарушенным принцип целостности абстракций.

Упражнение. Попробуйте найти эту неестественность, не читая пока дальше.

Вспомните, указатель сокращений действует так, что в точке, где он расположен, становятся непосредственно видимы имена из видимой части перечисленных в нем пакетов. Но (с учетом порядка построения области непосредственной видимости) не все такие имена, а только те, которые не закрыты именами, непосредственно видимыми в этой точке без указателя сокращений. Здесь самое главное – в том, что «не все». Обоснование у авторов, конечно, имеется, однако оказывается нарушенным важнейший принцип – принцип целостности.

Действительно, если пакет – логически связанная совокупность ресурсов, или, иными словами, модель мира, то указатель сокращений по основной идее, «вскрывая» пакет, позволяет удобно работать с этой моделью. Однако в том-то и дело, что пакет может быть вскрыт не полностью, и тогда целостность модели оказывается нарушенной.

Пример:

```
package P is                 -- модель мира:
   type T1 is range 1..10;   -- типы,
   ...
   type T10 is ...
   procedure P1 ...          --  процедуры,
   ...
   procedure P10 ...
   I1 : T1;                  -- переменные
   I10 : T10;
end P;
with P; use P; -- работа в модели мира P
procedure K is
-- нет объявлений имен T1-T10, P1-P10, I1-I10
begin
   I1:= 1;  -- I1 – компонента модели
   ...
   declare  -- блок, область локализации
      type T1 is range -10.. 10;
      I1 : INTEGER;
      ... -- работа не в модели
      use P;
      -- казалось бы, снова нужна модель P
      begin --  но I1 будет не то !!
         I1:= 1;  -- I1  не из модели, целостность модели нарушена!!
         I2:= 1;  -- I2 снова в модели !!
   ...
```

Упражнение. Постарайтесь найти доводы в пользу адовской семантики указателя сокращений. Ведь зачем-то она определена именно так!

Подсказка. Авторы отдавали приоритет явным объявлениям перед неявными (разумно, если не противоречит более важным принципам).

Еще более запутанные ситуации возможны при сочетании указателя сокращений с неявными объявлениями операций для производных типов. Ведь такие объявления равноправны с явными объявлениями!

Упражнение. Придумайте соответствующие ситуации (например, когда имя из модели мира становится недоступным из-за наследуемой операции, объявленной в весьма «отдаленном» пакете).

Итак, доказана неформальная теорема о свойствах аппарата именования в Аде: именование в Аде сложное и в определенных сочетаниях опасное, нарушающее важнейшие общеязыковые принципы. Вместе с тем этот аппарат по сложности и выразительной мощи в значительной степени согласован с аналогичными характеристиками языка в целом.

Глава 11

Обмен с внешней средой

11.1. Специфика обмена 220
11.2. Назначение и структура аппарата обмена 223
11.3. Файловая модель обмена в Аде 224
11.4. Программирование специальных устройств 233

11.1. Специфика обмена

До сих пор мы занимались внутренним миром исполнителя, игнорировали одну из важнейших технологических потребностей – потребность управлять обменом данными между исполнителем и внешней средой (управлять, как часто говорят, вводом-выводом данных). Настало время поговорить о связи исполнителя с внешним миром.

Конечно, можно считать, что такая связь обеспечивается соответствующими базисными операциями исполнителя; они имеются в каждом исполнителе (**почему?**). Управление обменом состоит в указании подходящей последовательности таких базисных операций. Казалось бы, никаких особых проблем.

На самом деле управление обменом обладает целым рядом особенностей, создающих специфические проблемы на всех уровнях проектирования – от создания аппаратуры до создания конкретных прикладных программ.

Источник всех таких проблем – в потенциально неограниченной сложности, изменчивости и непредсказуемости внешней среды, исключительном разнообразии требований к обмену с ее стороны. Коротко говоря, ***внутренний мир исполнителя обычно несравненно беднее и определеннее, чем его внешний мир***.

Конечно, проектировать связь исполнителя с внешним миром в каждом конкретном случае удается только за счет построения моделей внешнего мира, с нужной степенью подробности отражающих его особенности. Базовый ЯП должен содержать средства для построения таких моделей. Другими словами, в нем должны быть подходящие средства абстракции и конкретизации. Вместе с тем возможны и встроенные в язык готовые модели, которые авторы ЯП посчитали особо значимыми.

И то, и другое есть в Аде. Перечислим специфические особенности внешнего мира с точки зрения обмена, дадим общую характеристику соответствующего аппарата абстракции-конкретизации, а затем продемонстрируем его реализацию в Аде.

1. *Внешние объекты и их относительная независимость от исполнителя.* Первая особенность – в том, что объекты внешней среды и связи между ними, в отличие от внутренних объектов исполнителя, не находятся под его контролем полностью. Такие объекты могут возникать вне программы, взаимодействовать и изменяться, а также исчезать совершенно независимо от действий исполнителя. С точки зрения ЯП, это означает принципиальную невозможность полностью зафиксировать в языке смысл взаимодействия исполнителя с любым мыслимым внешним объектом. Для области встроенных систем, характеризуемой исключительным разнообразием внешних объектов, это означает необходимость иметь в базовом ЯП средства описания взаимодействия с объектами заранее неизвестной природы.

Например, в Аде это уже знакомые нам средства определения новых типов вместе с детальным управлением конкретным представлением объектов (вплоть до программирования в терминах другого ЯП).

Характерное проявление относительной независимости внешних объектов – невозможность определить в ЯП единый метод их именования. Поэтому в базо-

вом ЯП должен быть специальный способ представления внешних имен, причем подробности этого представления по необходимости должны зависеть от конкретной внешней среды. Например, в Аде внешнее имя представляется строкой символов, заключенной в кавычки. Детали зависят от конкретной среды (строго говоря, «определяются реализацией»).

2. *Разнообразие периферийных (внешних) устройств*. Вторая особенность тесно связана с первой и состоит в исключительном разнообразии и изменчивости набора устройств, выступающих во внешней среде партнерами исполнителя по обмену (взаимодействию). В сущности, такое разнообразие – прямое следствие все более широкого внедрения компьютеров в самые различные сферы деятельности.

Таким образом, с точки зрения набора устройств, внешний мир исполнителя изменяется весьма интенсивно. Причем меняется и в зависимости от рыночной конъюнктуры, и от прикладной области, и в зависимости от решаемой задачи, и от конкретных условий ее решения, и, наконец, динамически изменяется в процессе самого решения.

Например, зависимость от предметной области проявляется в том, что для числовых расчетов может оказаться достаточным ввода с перфокарт и вывода на печать; для игр нужно вводить с клавиатуры (или применять специальные манипуляторы) и выводить на экран (желательно цветной); для управления объектами в реальном времени нужно получать информацию от датчиков и выдавать управляющие сигналы органам управления объектами и т. п.

Зависимость от решаемой задачи проявляется, например, в том, что при подготовке книги к изданию с помощью компьютера первоначальный набор текста удобнее вести без применения специальных манипуляторов (типа «мышь», например), а собственно редактирование – с их помощью. Соответственно, вывод требуется на экран или принтеры различного качества и скорости или непосредственно на наборную машину.

Наконец, даже в процессе решения одной задачи внешняя среда может изменяться: приборы, датчики, органы управления, устройства ввода-вывода могут выходить из строя, конфигурация оборудования может изменяться по различным причинам.

Итак, будем считать обоснованным тезис об изменчивости внешней среды и (или) связи исполнителя со средой. Именно такого рода изменчивость создает особые сложности при программировании обмена и ставит проблему экономии (оптимизации) усилий создателя программы. Общий прием борьбы со сложностью такого рода нам хорошо знаком: нужен аппарат абстракции-конкретизации.

Современные средства программирования обмена обычно организованы так, что программист имеет возможность выбрать подходящий уровень абстракции при моделировании внешней среды своей программы. При этом удается в значительной степени игнорировать специфические особенности весьма разнообразных потенциальных внешних устройств. Так что содержательная часть программы пишется один раз, а настройка (конкретизация) на специфическое внешнее устройство выполняется в процессе связывания с этим устройством.

Однако процесс связывания при обмене с внешней средой имеет важную особенность. В общем случае в этом процессе невозможно обойтись без полного жизненного цикла программы, вплоть до проектирования специальных программ заново. Таким образом, аппарат связывания, ориентированный на обмен с внешним миром, должен содержать, как уже было сказано, развитые средства программирования.

3. *Человеческий фактор*. Третья особенность – в том, что в качестве источника и потребителя обмениваемых данных может выступать человек со своими специфическими свойствами и требованиями. Пример такого свойства – способность человека ошибаться при вводе данных. Пример требования – необходимость представлять данные в удобочитаемом или общепринятом при определенной деятельности виде.

С учетом того, что данные должны располагаться на ограниченном пространстве экрана, стандартном листе бумаги или чертеже и при этом создавать впечатляющий зрительный и (или) звуковой образ, возникает потребность в изощренных средствах форматирования, а также управления графикой, цветом, звуком и иными средствами воздействия на человека.

С этой точки зрения в Аде определены только простейшие возможности форматирования. Все остальное должно программироваться явно с применением средств развития.

4. *Динамизм и относительная ненадежность*. Четвертая особенность – динамизм внешних объектов. Из-за относительной независимости поведения внешних объектов достаточно полный статический (при трансляции программы) контроль их поведения невозможен.

Например, невозможно гарантировать правильность работы человека с клавиатурой или сохранность файла на внешнем носителе. В случае внутренних объектов статический контроль возможен именно за счет того, что все поведение такого объекта находится под полным контролем программы. Скажем, у целой переменной не может быть вещественного значения, потому что нет способа в программе выполнить подобное присваивание. Но никакими ограничениями ЯП нельзя запретить человеку ошибаться или испортить внешний носитель.

Поэтому при управлении обменом с внешней средой совершенно необходимым оказывается динамический контроль с помощью аппарата исключений. Итак, динамизм сочетается с ненадежностью внешних объектов.

5. *Параллелизм*. Пятая особенность – существенная асинхронность поведения внешних объектов по отношению к исполнителю. Это, конечно, также прямое следствие их относительной независимости и разнообразия. Исторически именно различия скорости центрального процессора и внешних устройств привели к изобретению аппарата прерываний и других средств программирования асинхронных процессов. Стремление к рациональному, эффективному и естественному взаимодействию с внешней средой, где могут относительно самостоятельно существовать активные объекты (устройства) со своим «жизненным ритмом», приводит к применению в области обмена с внешней средой аппарата управления асинхронными процессами.

Итак, завершая знакомство со специфическими особенностями обмена, подчеркнем, что в этой области требуется совместно использовать практически весь спектр изученных нами концепций и языковых конструктов – и моделирование, и связывание, и средства развития, и типы, и управление представлением, и исключения, и асинхронные процессы. А также родовые сегменты и перекрытия, как мы скоро увидим. С этой точки зрения рассматриваемая тема достойно завершает общее знакомство с базовым языком индустриального программирования, давая возможность продемонстрировать его характерные свойства в совокупности. Для Ады это естественно. Ведь программирование обмена – это в общем случае и есть программирование исполнителя как системы, встроенной в окружающую среду (то есть объемлющую систему), работающую в реальном масштабе времени.

11.2. Назначение и структура аппарата обмена

В соответствии с принципом обозначения повторяющегося специфика обмена оправдывает появление в развитых ЯП специализированного аппарата обмена. Этот аппарат предназначен для удовлетворения указанных выше потребностей (именование внешних объектов, связывание внешних объектов с внутренними, контроль и управление исключениями, форматирование, подключение устройств заранее неизвестной природы).

Аппарат обмена в традиционных ЯП обычно строится так, чтобы максимально освободить программиста от особенностей конкретных устройств ввода-вывода. Как правило, ЯП содержит достаточно абстрактную машинно независимую модель обмена, которая и поступает в распоряжение программиста. Все проблемы связывания (конкретизации обмена с учетом реальной внешней среды) решаются вне программы внеязыковыми средствами. Обычно это делается средствами операционной системы. При этом подключение совершенно новых устройств требует программирования соответствующих связывающих программ-драйверов, как правило, на уровне машинного языка. И выполняется оно не прикладными, а системными программистами.

В абстрактной модели обмена обычно пользуются понятием «логического внешнего устройства». Другими словами, это абстрактное устройство, отражающее существенные особенности реальных физических устройств некоторого класса и играющее роль модели таких устройств.

Важнейшая с точки зрения обмена особенность устройства ввода-вывода состоит в том, что к нему можно применять операции передачи и (или) получения данных определенного типа. Естественно, именно эта особенность отражена во всех логических устройствах обмена. Более тонкие особенности модели определяют, можно ли получить ранее переданные данные, как связан порядок передачи с порядком получения, какого рода контроль сопутствует обмену и т. п.

11.2.1. Файловая модель

Одна из наиболее распространенных моделей обмена – файловая модель. В ней внешние устройства представлены **файлами**. Файлы естественно считать именованными объектами некоторого предопределенного типа (ограниченного приватного в смысле Ады) с подходящим набором операций. От других моделей файловая отличается **независимостью** файлов.

Если специально не оговорено обратное, то данные из различных файлов никак не связаны между собой. Другими словами, из файла невозможно получить данное, переданное в другой файл.

Абстрактные файлы называют также потоками, каналами, наборами, фондами, теками (иногда связывая с этими названиями определенную специфику).

Наиболее распространенные файловые модели – последовательная и индексно-последовательная, соответствующие реальным устройствам последовательного и прямого доступа.

Примерами устройств последовательного доступа служат магнитофоны (магнитные ленты), прямого доступа – дисководы (магнитные диски). Характерная особенность последовательного файла – возможность получать данные только в том порядке, в котором они были ранее переданы. Особенность индексно-последовательного файла – возможность произвольно менять этот порядок, управляя позицией, в которой выполняется обмен. Эта позиция однозначно определяется индексом (аналогом адреса внутренней памяти).

В качестве примера рассмотрим абстрактную модель обмена в Аде. Соответствующий аппарат обмена называют аппаратом обмена высокого уровня. Название связано с «высоким уровнем» абстракции соответствующей модели. Он проявляется в практически полном отсутствии в этой модели специфических особенностей реальных устройств обмена (нет никаких зон МЛ, дорожек или цилиндров МД и т. п.).

В Аде имеется и аппарат обмена низкого уровня. Его зависимость от реальных устройств проявляется, в частности, в том, что программист обязан полностью определять организацию связи между последовательно передаваемыми или получаемыми данными (в соответствии с назначением и индивидуальными особенностями реального устройства). Априори не предполагается никакого аналога сохранения этих данных в именованных файлах.

11.3. Файловая модель обмена в Аде

Файловая модель представлена в Аде четырьмя предопределенными родовыми пакетами: последовательный_обмен, прямой_обмен, текстовый_обмен и исключения_обмена. Приведем в качестве примера спецификацию одного из этих пакетов. Подробнее со средствами обмена в Аде можно познакомиться в [17].

11.3.1. Последовательный обмен

```
with исключения_обмена;
generic
   type тип_элемента is private;
package последовательный_обмен is
   type файловый is limited private;
   type режим_обмена is(ввод, вывод);  -- управление файлами
procedure создать(файл :in out файловый; -- внутренний файл
                      режим : in режим_обмена;
                      имя   : in строчный := " ";  -- внешнее
                      доступ: in строчный := " "); -- правила доступа,
                                                   -- физическая организация
procedure открыть(файл :in out файловый;
                режим : in режим_обмена;
                имя : in строчный;
                доступ: in строчный := " ");
procedure закрыть(файл :in out файловый);
procedure удалить(файл :in out файловый);
procedure сначала(файл :in out файловый;
                режим : in режим_обмена);
procedure сначала(файл :in out файловый);
function режим(файл : in файловый) return режим_обмена;
function имя(файл : in файловый) return строчный;
function доступ(файл : in файловый) return строчный;
function открыт(файл : in файловый) return BOOLEAN;

-- операции собственно обмена
procedure читать(файл :in файловый;
                      элемент : out тип_элемента);

procedure писать(файл :in файловый; элемент : out тип_элемента);

function конец_файла(файл : in файловый) return BOOLEAN;

-- исключения
статус_неправильный : exception renames исключения_обмена.статус_неправильный;
-- файл не открыт или попытка открыть неоткрытый файл
режим_неправильный : exception renames исключения_обмена.режим_неправильный;
-- ввод из выводного или наоборот
имя_неправильное : exception renames исключения_обмена.имя_неправильное;
-- очевидно
использование_неправильное : exception renames исключения_обмена.использование_не-
правильное;
-- попытка создать входной с доступом выходного и т. п.
устройство_неисправно : exception renames исключения_обмена.устройство_неисправно;
```

```
-- отказ соответствующего внешнего устройства, не позволяющий завершить операцию
-- обмена
закончен_файл : exception renames исключения_обмена.закончен_файл;
-- попытка прочитать маркер конца файла
данные_неправильные : exception renames исключения_обмена.данные_неправильные;
-- данные нельзя интерпретировать в соответствии с указанным типом элемента.
private
-- определяется реализацией языка
end последовательный_обмен;
```

Представлено формализованное описание абстрактной модели последовательного обмена, принятой в Аде. В качестве метаязыка мы воспользовались самим языком Ада. Спецификацию пакета можно рассматривать в качестве синтаксической части такого формализованного описания. Она полностью определяет строение вызовов операций и объявлений данных, допустимых в некотором специализированном языке обмена. Семантика этих операций и объявлений окончательно фиксируется отсутствующими частями (в том числе телами) пакетов. Для пользователя она становится известной из комментариев и специально для него предназначенной документации.

Итак, мы одновременно продемонстрировали метод и результат описания посредством Ады реального специализированного (проблемно-ориентированного) языка (в другой терминологии – пакета прикладных программ) для управления обменом. Таким методом можно описывать и проблемно-ориентированные языки (прикладные пакеты), которые не предполагается полностью (или даже частично) реализовывать на Аде. Кстати, именно таков наш язык управления обменом (для реализации предопределенных пакетов придется воспользоваться всеми возможностями инструментальной и целевой машин). Использование Ады в качестве языка спецификаций в настоящее время довольно широко распространено.

11.3.2. Комментарий

Итак, в контексте пакета «последовательный обмен» внешний мир исполнителя представлен совокупностью внешних файлов, идентифицируемых по уникальным именам-строкам. Дополнительные свойства внешних файлов (возможность только вводить, только выводить, данные о разметке и т. п.) указываются в специальной строке «доступ». Таким образом, внешние файлы – это абстрактные внешние устройства.

Во внутреннем мире исполнителя внешние файлы представлены внутренними объектами ограниченного приватного типа «файловый» с набором операций и исключений, зафиксированным в определяющем пакете последовательный_обмен. После соответствующей конкретизации (настройки) этого пакета на нужный тип вводимых (или) выводимых данных (тип_элемента) в контексте этого пакета можно:

1) объявлять внутренние файлы:

```
A,B : файловый;
```

2) создавать внешние файлы и связывать их с объявленными внутренними:

```
создать(A,вывод,"пример","последовательный");
```

При этом правила указания имени и доступа зависят от конкретной внешней среды («определяются реализацией»);

3) открывать ранее созданные внешние файлы, связывая их с внутренними:

```
открыть(A,ввод,"пример","последовательный");
```

Ясно, что открывать для ввода имеет смысл только такие внешние файлы, в которые ранее что-то уже выводилось, либо файлы, которым в реальной внешней среде соответствуют источники данных (клавиатура, устройство ввода с перфокарт и т. п.);

4) закрывать файлы, разрывая связь внутреннего файла с внешним:

```
закрыть(A);
```

Операция открытия может потребовать установки подходящего диска или кассеты с нужным внешним файлом на подходящее устройство обмена. Операция закрытия позволяет освободить это устройство для других целей;

5) удалять файлы из внешней среды, делая их впредь недоступными:

```
удалить(A);
```

Этой операцией следует пользоваться очень осторожно;

6) установить файл в начальную позицию.
Позиции линейно упорядочены начиная с 1. Операция чтения или записи увеличивает позицию на 1 (после своего выполнения). Понятие позиции касается внутренних, а не внешних файлов. Допустимо связывать с одним внешним файлом несколько внутренних. Они могут быть разных режимов и могут находиться в разных позициях. Однако это возможно не во всякой среде;

7) узнать режим обмена, внешнее имя, характеристику доступа, а также узнать, открыт ли файл;

8) наконец, можно прочитать или записать объект данных нужного типа.
Например, если объявлен тип «таблица», то после конкретизации

```
package обмен_таблиц is new последовательный_обмен(таблица);
use обмен_таблиц;
T : таблица;
```

можно объявить

```
A : файловый; ...
открыть(A, вывод, "таблицы","последовательный");
loop -- формирование таблицы
   писать(A, T);
end loop;
закрыть(A);
```

Затем в аналогичном контексте можно прочитать сформированный ранее файл таблиц:

```
открыть(A,ввод, "таблицы", "последовательный");
if not конец_файла(A) then читать(A, T);...
закрыть(A);
```

Тем самым показано и применение функции «конец_файла». Смысл исключений указан в комментариях определяющего пакета.

Вопрос. Зачем нужен отдельный пакет исключений обмена, а также переименования в родовом пакете последовательный_обмен? Почему нельзя просто объявить исключения в этом родовом пакете?

Ответ. Такой прием описания позволяет организовать единообразную реакцию на исключения обмена. Если бы исключения не были переобъявлены, то они не были бы видимы в контексте, где применяется пакет последовательный_обмен (они были бы видимы только в самом этом пакете). Поэтому нельзя было бы в этом контексте задавать реакцию на эти исключения. А если объявить исключения в родовом пакете, то для каждой конкретизации этого пакета они были бы своими (другими) и при совместном применении различных конкретизаций (для обмена данных различных типов) попытка задать реакцию на исключения приводила бы к неудобствам или конфликту наименований.

Доказана *неформальная теорема*: **исключения обмена рационально объявлять в предопределенном пакете и переименовывать в родовых специализированных пакетах**.

Вопрос. Почему реализация обмена родовая? Почему нельзя в одном пакете определять обмен данных различных типов?

Ответ. В соответствии с концепцией уникальности типа процедуры обмена должны иметь точную спецификацию параметров. Тем самым фиксируется тип обмениваемых данных. Отсюда следует и фундаментальное свойство адовских файлов – однородность; каждый файл характеризуется единым типом элементов (все элементы файла – одного типа!).

Доказана еще одна *неформальная теорема*: **концепция уникальности типа влечет однородность файлов**.

Представление внутренних объектов во внешних файлах в общем случае в Аде не определено. Более того, оно не обязано быть зафиксированным в каких-либо документах, доступных программисту. Это представление «зависит от реализации», но не «определяется реализацией» (последнее означает, что свойство обязано быть описано в документации для программистов).
Поэтому вводить можно только то, что ранее было выведено с помощью пакета для того же типа данных. Другими словами, **последовательный (и прямой) обмен «неполноценен» в том отношении, что создавать и потреблять внешние данные при таком обмене невозможно без компьютера**.
Это снова неформальная теорема. Она не распространяется на текстовый обмен, при котором можно вводить любые данные, представленные в соответствии с синтаксисом ЯП Ада. Именно синтаксис и служит документом, фиксирующим в этом случае правила представления внутренних объектов во внешних файлах. При текстовом обмене допустимы и некоторые дополнительные возможности структуризации текстов (форматирования), выходящие за рамки синтаксиса Ады (разбиение на строчки, страницы и т. п.).

В пакете прямой_обмен предоставлена возможность управлять позицией (индексом) обмена в операциях «читать» и «писать», а также устанавливать и узна-

вать текущую позицию с помощью операций установить_индекс и дай_индекс. Функция «размер» позволяет узнать максимальное значение индекса, по которому производилась запись в данный файл (то есть узнать число элементов во внешнем файле).

Последовательный и прямой обмены учитывают относительную независимость внешних объектов, их динамизм и (частично) разнообразие внешних устройств. Однако совершенно не учитывают человеческого фактора. В сущности, последовательный и прямой обмены предназначены для взаимосвязи с устройствами внешней памяти (магнитными лентами, магнитными дисками и т. п.) и не предназначены для взаимодействия с человеком или устройствами, которые служат не для хранения данных (датчики, органы управления и т. п.).

Аппаратом, явно учитывающим человеческий фактор, в Аде служит предопределенный пакет текстовый_обмен. Приведем только его структуру, которая нам понадобится в дальнейшем:

```
with исключения_обмена;
package текстовый_обмен is -- это не родовой пакет!
... -- далее идут вложенные родовые пакеты
   generic -- Родовой пакет для обмена значений целых типов
      type число is range < >;
   package целочисленный_обмен is ...
   generic -- Родовые пакеты для обмена вещественных
      type число is digits < >;
   package плавающий_обмен is ...
   generic
      type число is delta < >;
   package фиксированный_обмен is ...
   generic -- Родовой пакет для обмена перечисляемых типов.
      type перечисляемый is(< >);
   package перечисляемый_обмен is ...
exception
... -- исключения (как в последовательном обмене плюс одно дополнительное "нет_мес-
та") private ... -- определяется реализацией end текстовый_обмен;
```

11.3.3. Пример обмена. Программа диалога

Постановка задачи. Следует организовать диалог с пользователем системы, хранящей сведения о товарах (например, автомобилях), имеющихся в продаже.

Обратите внимание: это совершенно новый вид задачи. Мы уже программировали алгоритм вычисления некоторой функции – задача ставилась в форме спецификации требуемой функции. Программировали совокупность модулей, предоставляющую комплекс программных услуг, – задача ставилась в форме спецификации перечня услуг. Теперь нужно организовать диалог. Это не функция и не комплекс услуг – это взаимодействие.

Удобной формой спецификации взаимодействия служит сценарий. Другими словами, это описание ролей партнеров по взаимодействию (описание их поведе-

ния с учетом возможного поведения партнера). Отличие от обычного театрального сценария – в том, что в общем случае последовательность действий партнеров не фиксируется.

Вопрос. В чем отличие сценария от комплекса услуг?

Таким образом, при решении диалоговых задач начинать проектирование следует с разработки сценария как исходной «функциональной» спецификации задачи, а затем продолжать решение обычной детализацией.

Сценарий нашего диалога прост.

Система. Начинает диалог, предлагая пользователю выбрать желательный цвет (автомобиля).

Пользователь. Отвечает, печатая название цвета (тем самым запрашивая автомобиль указанного цвета).

Система. В ответ на запрос сообщает число автомобилей нужного цвета, имеющихся в продаже, либо указывает на ошибку в запросе и предлагает повторить попытку.

Пример диалога (ответы пользователя – справа от двоеточия)
Выберите цвет: Черный.
Недопустимый цвет, попытаемся еще раз.
Выберите цвет: Голубой. Голубой цвет: 173.
Выберите цвет: Желтый. Желтый цвет: 10.

Программа диалога. Приведем вариант программы диалога:

```
with текстовый_обмен; use текстовый_обмен;
procedure диалог is
   type цвет is(белый, красный, оранжевый, желтый,
      зеленый, голубой, коричневый);
   таблица: array(цвет) of INTEGER:=
      (20,17,43,10,28,173,87);
   выбранный_цвет : цвет;
   package для_цвета is new перечисляемый_обмен (цвет);
   package для_чисел is new целочисленный_обмен (INTEGER);

use для_цвета, для_чисел;
begin
      loop
         declare -- блок нужен для размещения реакции на исключение
               -- ввод цвета:
               послать("Выберите цвет:");
               получить(выбранный_цвет);
               -- конец ввода цвета

               -- вывод ответа:
               установить_колонку (5);
                  -- отступ – 5 позиций
               послать(выбранный_цвет);
```

```
                послать("цвет:");
                установить_колонку (40);
                   -- чтобы выделялось количество автомобилей,
                послать(таблица(выбранный_цвет), 4);
                   -- размер поля в 4 позиции достаточен
                   -- для чисел из таблицы
                новая_строчка; -- конец вывода ответа

          exception
             -- реакция на ошибки пользователя
             when данные_неправильные =>
                послать("Недопустимый цвет. Еще раз.");
                новая_строчка(2);
             end; -- конец блока (и реакции на ошибку)
   end loop;
end диалог;
```

Использованы подразумеваемые внешние файлы. Обычно это клавиатура для ввода и экран для вывода. Точнее, управлять назначением устройств ввода-вывода в рамках абстрактной модели невозможно. Требуются данные (а возможно, и операции), описываемые в конкретной реализации ЯП.

11.3.4. Отступление о видимости и родовых пакетах

Мы уже отмечали ряд неприятных свойств аппарата управления видимостью в Аде. Покажем еще один пример, когда он проявляет себя нелучшим образом.

Допустим, что пользоваться обменом для нескольких конкретных типов (например, чисел, цветов, строк) приходится часто, и возникает идея создать подходящий пакет, полностью обеспечивающий нужный контекст. Слово «полностью» подчеркивает естественное требование, чтобы для работы в нужном контексте пользователю было достаточно написать указатель контекста с одним только именем такого пакета. Пусть для определенности нужны именно те процедуры, которыми мы воспользовались при программировании диалога. Казалось бы, достаточно написать пакет

```
with текстовый_обмен; use текстовый_обмен;
package обмен_чисел_цветов_строк is
   type цвет is (белый, ... .коричневый);
   package для_цвета is new перечисляемый_обмен(цвет);
   package для_чисел is new целочисленный_обмен(INTEGER);
   use для_цвета; для_чисел;
end обмен_чисел_цветов_строк;
```

Так что процедуру, аналогичную нашей процедуре «диалог», можно начинать так:

```
with обмен_чисел_цветов_строк; use обмен_чисел_цветов_строк;
procedure новый_диалог is ...;
```

Однако правила видимости в Аде не позволяют так работать. Ведь в процедуре «новый_диалог» будет непосредственно видимо лишь то, что объявлено в пакете из указателя сокращений use. Так что нужные процедуры придется указывать полным именем. Например:

```
для_чисел.послать(...);
для_цветов.получить(...); и т. п.
```

Таким образом, пользователь вынужден знать не только имя пакета, но и его содержимое, да к тому же вынужден применять длинные имена. Это явно не входило в наши планы. Другими словами, мы не достигли нужного уровня абстракции предоставляемой услуги.

Чтобы его достичь, необходимо в пакете для_чисел_цветов_строк применить переименование всех нужных процедур из предопределенных пакетов. Например:

```
procedure послать(элемент : out цвет;
   поле : in размер_поля := для_цвета.подразумеваемое_поле;
   нижний : in BOOLEAN := для_цвета.подразумеваемый_нижний)
      renames для_цвета.послать;
```

И так для всех (!) нужных процедур.

Назовем отмеченную неприятность **проблемой транзита импортированных имен** (ведь суть проблемы – в передаче заимствованных коротких имен через промежуточный модуль). Эта проблема характерна и для других ЯП с развитым управлением контекстом (например, для Модулы-2, которой мы еще займемся). В Аде она решена (в отличие от той же Модулы-2), хотя и с весомыми накладными расходами.

Упражнение (повышенной трудности). Предложите и обоснуйте решение проблемы транзита.

Итак, абстрактная модель Ады характеризуется:

1) понятием файла, разграничением внешних и внутренних файлов и предопределенным типом «файловый»;
2) развитым аппаратом связывания с файлами подходящих наборов операций, представленным совокупностью пакетов с родовыми параметрами;
3) однородностью файлов;
4) полиморфизмом и вариаргументностью предоставляемых операций (вариаргументность – это возможность применять процедуры с переменным числом аргументов за счет подразумеваемых значений опущенных аргументов);
5) форматированием, встроенным непосредственно в операции управления обменом (отсутствует, например, аналог именованного формата в Фортране).

Упражнение. Придумайте варианты развития описанного нами диалога и реализуйте его. В случаях, когда возникают сомнения в точной семантике операций, самостоятельно опишите (и обоснуйте) нужное вам уточнение эффекта операции или воспользуйтесь литературой по Аде.

11.4. Программирование специальных устройств

Чтобы управлять специальным внешним устройством, необходимо иметь соответствующую аппаратуру – само устройство и аппаратный адаптер, связывающий подключаемое устройство с основным исполнителем. При этих условиях новое устройство становится источником (и потребителем) определенных воздействий на исполнитель (со стороны исполнителя). Так как обычно внешнее устройство работает асинхронно с основным исполнителем, целесообразно рассматривать его как задачу, с которой можно взаимодействовать посредством аппаратных прерываний, специальных регистров, выделенных адресов и т. п. Назовем такую задачу **аппаратной**.

Изменить принцип функционирования аппаратной задачи (перепрограммировать ее) часто практически невозможно или нецелесообразно. Будем считать его фиксированным. С другой стороны, с точки зрения создаваемого комплекса программ, возможности обмена, предоставляемые аппаратной задачей, обычно выглядят как возможности излишне низкого уровня (абстракции). Они детально отражают поведение устройства и обычно весьма далеки от планируемых его содержательных функций в создаваемой системе.

Единственный способ построить более содержательное (и вместе с тем более абстрактное) устройство – создать программную модель устройства в виде асинхронного процесса, взаимодействующего с аппаратной задачей. Такая программная модель называется **драйвером устройства**.

При создании драйвера естественно моделировать аппаратные регистры и адреса данными подходящих типов, использовать спецификацию представления для привязки этих данных к конкретным компонентам аппаратуры, а аппаратные прерывания использовать для синхронизации драйвера с аппаратной задачей, связывая входы драйвера с адресами соответствующих аппаратных прерываний.

Итак, общий метод программирования специальных внешних устройств сводится к построению подходящего драйвера. При этом оказываются полезными развитые типы данных и спецификации их представления.

Важно понимать, что специфицировать представление приходится не только для привязки к адресам и структуре регистров. Например, перечисляемые типы удобны в модели-драйвере, но их обычно аппаратура не воспринимает. Поэтому часть работы, в обычных условиях «отдаваемой на откуп» компилятору, приходится делать вручную, выписывая явно конкретные коды для значений перечисляемого типа. Например:

```
type защита is(обычная, ограниченная, строго_ограниченная,
   секретная, совершенно_секретная);
for защита use(обычная => 0, ограниченная => 1,
   строго_ограниченная => 2, секретная => 4,
   совершенно_секретная => 8);
```

В результате гарантируется, что драйвер передаст аппаратуре вполне определенные числовые коды, предусмотренные ее конструкцией.

Рассмотрим пример драйвера из [17]. Символ, вводимый с клавиатуры, вырабатывает прерывание с адресом 8#100# и выдачей соответствующего символа в буферный регистр. (Человек за клавиатурой играет роль аппаратной задачи.)

Напишем драйвер, который сохраняет введенный символ в локальной переменной, доступной из любой обслуживаемой задачи по входу «взять_символ»:

```
task драйвер_клавиатуры is
   entry взять_символ(симв : out символьный);
   entry есть_символ; -- аппаратное прерывание
   for есть_символ use at 8#100#;
end драйвер_клавиатуры;
-- драйвер позволяет обслуживаемой задаче быть независимой
-- от конкретных адресов и прерываний и в этом смысле
-- служит абстрактной моделью клавиатуры.

task body драйвер_клавиатуры is
   символ : символьный; -- рабочая переменная
   буф_регистр : символьный;
   for буф_регистр use at 8#177462#;
   -- так элементы аппаратуры представляются данными адовских типов
begin
   loop
      accept есть_символ do символ := буф_регистр end есть_символ;
      accept взять_символ(симв : out символьный) do
         симв := символ;
      end взять_символ;
   end loop;
end драйвер_клавиатуры;
```

Достигнув в цикле оператора приема «есть_символ», драйвер ждет аппаратного прерывания. Предварительно он обязан разрешить его и заслать в вектор адресов прерываний ссылку на тело оператора приема этого входа. Делать все это обязана реализация оператора приема аппаратного входа в Аде. Отличить аппаратный вход можно по спецификации его представления.

После прерывания тело оператора приема выполняется (происходит аппаратно-программное рандеву). И драйвер готов обслужить машинно независимую задачу по входу «взять_символ». Здесь рандеву самое обычное. Далее все повторяется в бесконечном цикле.

Итак, при программировании специального устройства потребовалось:

- построить модель аппаратной задачи (в нашем случае она представлена переменной буф_регистр и входом есть_символ);
- связать эту модель с конкретной аппаратурой (в нашем случае – две спецификации представления);
- построить на основе аппаратной модели содержательную модель устройства – задачу-драйвер.

Теперь можно пользоваться драйвером в качестве специального устройства обмена в прикладных программах.

Упражнение. Разберите примеры управления специальными устройствами в книгах Пайла [17] и Янга [18]. Убедитесь, что в них явно или неявно присутствуют и аппаратная модель, и связывание, и драйвер.

На этом закончим построение модели А (а тем самым изучение основных абстракций современного индустриального ЯП с технологической позиции) – изученные аспекты Ады и составили модель А.

Глава 12

Два альтернативных принципа создания ЯП

12.1. Принцип сундука 238
12.2. Закон распространения сложности ЯП 238
12.3. Принцип чемоданчика 239
12.4. Обзор языка Модула-2 239
12.5. Пример М-программы 241
12.6. Языковая ниша 248
12.7. Принцип чемоданчика в проектных решениях ЯП Модула-2 249
12.8. Принцип чайника 257
12.9. ЯП Оберон 258

12.1. Принцип сундука

Построив модель А, мы достигли переломного момента книги. До сих пор накапливалось знание о технологических потребностях и удовлетворяющих эти потребности языковых конструктах. Теперь накоплено достаточно материала, чтобы взглянуть на него критически, с позиции автора современного ЯП.

И раньше мы не забывали об авторской позиции, выделяли концепции и конструкты, помогающие создавать ЯП, анализировать и оценивать его. Однако эти принципы и концепции, как правило, касались отдельных групп конструктов. Наша ближайшая крупная цель – указать на два принципа, касающихся ЯП в целом, и обсудить их влияние на современное языкотворчество.

Для краткости и выразительности дадим им названия «принцип сундука» и «принцип чемоданчика».

На примере Ады мы видели, как выявляемые технологические потребности приводили к новым конструктам. Может показаться, что на этом пути будут получаться все более высококачественные ЯП. К сожалению, большинство современных индустриальных ЯП носят на себе родимые пятна такого примитивного критерия качества. Это характерно и для Кобола, и для ПЛ/1, и для Фортрана-77, и для Ады.

Основной принцип конструирования, которым руководствовались авторы этих ЯП, в упрощенной форме можно сформулировать так: для каждой значимой в ПО технологической потребности в языке должно быть готовое выразительное средство. Короче: каждой значимой потребности – готовый конструкт. Этот принцип заготовленности конструктов и назовем **принципом сундука** (именно в сундуках хранят много всякого на всякий случай).

Как показывает опыт, безудержное применение принципа сундука ведет к громоздким, сложным, дорогим в реализации, обучении и использовании языкам-монстрам с тяжеловесным базисом и несбалансированными средствами развития. Сундук и есть сундук!

12.2. Закон распространения сложности ЯП

Бывают взаимодействия, сложность которых по существу не зависит от собственной сложности взаимодействующих объектов. Например, процесс и результат столкновения человека с автомобилем в первом приближении никак не связаны со сложностью человека и автомобиля. Сложность вызова процедуры непосредственно не связана с ее внутренней сложностью и сложностью вызывающего контекста. В подобных ситуациях сложность инкапсулирована. Образно говоря, простота взаимодействия обеспечивается «небольшой площадью» взаимодействия потенциально весьма сложных объектов.

С другой стороны, язык программирования сам служит «поверхностью» взаимодействия авторов, реализаторов, преподавателей и пользователей ЯП.

Такая специфическая роль ЯП определяет справедливость для него следующего **закона распространения сложности**: собственная сложность ЯП распространяется на все аспекты его «жизни» (описание, реализацию, использование, обучение и т. д.). Никлаус Вирт отмечает частный случай этого закона [19] как самое главное, что следует усвоить о реализации ЯП.

> Н. Вирт – один из самых авторитетных специалистов по ЯП, лауреат премии Тьюринга за создание таких известных ЯП, как Алгол W, Паскаль, Модула, Модула-2.

Итак, ориентация на принцип сундука повышает собственную сложность ЯП, что по закону распространения сложности приводит к росту сложности его освоения (которая, в свою очередь, может оказаться катастрофической для его судьбы – достаточно сопоставить судьбу Паскаля с судьбой Алгола-68).

Вспомним, однако, что основной принятый нами критерий качества базового ЯП – его способность снижать сложность, помогать в борьбе с основной проблемой программирования. Налицо тупик, в который ведет принцип сундука. Вирту принадлежит принцип, указывающий выход из этого тупика.

12.3. Принцип чемоданчика

Н. Вирт неоднократно отмечал, что **самое трудное при создании ЯП – решить, от чего следует отказаться**. Объясняя принципы конструирования своего (теперь уже предпоследнего) ЯП Модула-2 (поразившего специалистов элегантностью), Вирт развил эту идею и сформулировал следующий **принцип языкового минимума**: **в ЯП следует включать только такие концепции и конструкты, без которых совершенно невозможно обойтись**.

Назовем этот принцип минимума **принципом чемоданчика** по контрасту с принципом сундука (в чемоданчик кладут только абсолютно необходимое).

Продемонстрируем этот важнейший метаязыковый принцип на примере конкретных решений, принятых при создании ЯП Модула-2, и сопоставим их с решениями, воплощенными в Аде. Но сначала придется в общих чертах познакомиться с Модулой-2.

12.4. Обзор языка Модула-2

Язык Модула-2, созданный в конце 70-х годов, прямой наследник ЯП Паскаль и Модула, созданных Н. Виртом в конце 60-х и середине 70-х годов соответственно. Это ЯП общего назначения, ориентированный на относительно скромные ресурсы как инструментальной, так и целевой машины. Автор предназначал его для небольших, в том числе персональных, компьютеров («для небольшой однопроцессорной ЭВМ»).

В Модуле-2 автору удалось соединить простоту и естественность основных конструктов Паскаля с мощными и изящными средствами модуляризации. Короче говоря, Модула-2 – это модульный Паскаль.

Специалисты обратили внимание на очередное достижение Вирта с первых публикаций. В настоящее время интерес к Модуле-2 становится всеобщим, так как появились высококачественные его реализации. Замечательная особенность ЯП, обеспечившая его достоинства, – следование принципу чемоданчика.

12.4.1. Характеристика Модулы-2 в координатах фон-неймановского языкового пространства (технологическая позиция)

По сравнению с моделью А отсутствуют производные типы; концепция типа ориентирована скорее на структуру, чем на имена; существенно меньше предопределенных типов; резко ограничен аппарат управления асинхронными процессами; сильно упрощены управление видимостью и аппарат раздельной компиляции (в частности, отсутствуют вторичные модули); отсутствует аппарат управления точностью расчетов; упрощена предопределенная модель обмена; отсутствуют родовые модули; резко ограничено управление представлением; отсутствует управление исключениями.

С другой стороны, с точки зрения языкового пространства Модула-2 ничего не добавляет к модели А. Иначе говоря, верна **неформальная теорема**: **для всякого конструкта Модулы-2 в Аде найдется конструкт с аналогичными возможностями**, то есть Модулу-2 можно назвать технологическим подмножеством Ады.

12.4.2. Характеристика Модулы-2 в терминах концептуальной схемы

Для краткости свойства Модулы-2 назовем М-свойствами, а свойства Ады (А-модели) – А-свойствами.

Базис. Скалярная М-сигнатура – точное подмножество А-сигнатуры. Отсутствуют фиксированные вещественные типы и вообще управление точностью. Но имеются целые, вещественные, символьные, логические типы (с более или менее устоявшимся набором операций).

Структурная М-сигнатура содержит регулярные, комбинированные и ссылочные типы, аналогично А-сигнатуре. Но в ней имеются также процедурные и множественные типы. Правда, весьма ограниченные.

Именно значением процедурного типа не может быть стандартная процедура или процедура, вложенная в другую процедуру. А «множественный» тип (SET OF T), класс значений которого состоит из всех возможных множеств значений исходного типа T, можно образовывать только для исходных типов малой мощности (чтобы множество можно было представить одним машинным словом – двоичной шкалой; это пример влияния на язык реализаторской позиции).

Имеются обычные управляющие структуры (последовательности, развилки, циклы, процедуры) и ограниченные средства управления асинхронными процес-

сами. При этом настоящий параллелизм подразумевается только для так называемых периферийных процессов (аппаратных задач, соответствующих внешним устройствам).

Развитие. Можно создавать операционные абстракции (процедуры и функции) и абстракции данных (именованные типы с возможностью ограничивать набор применимых операций – аналог приватных А-типов). Основное средство развития – модуль. Это аналог А-пакета.

Защита. Обязательные объявления с соответствующим контролем поведения. Подразумеваются статический контроль так называемой совместимости типов и динамический контроль за допустимостью значений переменных. Роль подтипа по существу играет тип, объявленный как отрезок другого типа. Отрезки одного исходного типа совместимы между собой и с исходным типом. Формально понятие подтипа отсутствует. Производные типы отсутствуют (и формально, и по существу).

> Понятие типа и совместимости типов в авторском описании определено нечетко. Не ясно, какие типы равны. Можно подозревать, что равными считаются типы, названные одинаковыми именами. Так как нет производных типов, то операции явно с типом не связаны (нет проблемы неявного определения операций для производных типов).

Аппарат исключений не предусмотрен.

Мощное средство прогнозирования и контроля фактически предоставляет аппарат управления видимостью – списки экспортируемых и импортируемых имен. С их помощью обеспечивается инкапсуляция в рамках модулей (аналогов А-пакетов).

Исполнитель. Характеризуется набором предопределенных и определяемых реализацией модулей, в совокупности обеспечивающих рациональное для небольших компьютеров сочетание общеязыковых (резидентных, постоянно присутствующих в компиляторе и (или) целевой машине) и специализированных средств. Среди последних – аппарат файлового обмена, управления параллельными процессами посредством сигналов, управления динамическим распределением памяти.

Архитектура. Характеризуется принципом чемоданчика. Именно поэтому мы особенно интересуемся Модулой-2.

12.5. Пример М-программы

Рассмотрим решение уже известной нам задачи об управлении сетями. Цель – создание у читателя «зрительного образа» М-программ, а также подробное знакомство с теми свойствами ЯП, которые помогут продемонстрировать принцип чемоданчика. Требования к реализации комплекса услуг по управлению сетями те же, что и в А-случае (надежность, целостность, модифицируемость).

Однако в самом начале следует сказать о ключевом понятии Модулы-2. Название этого понятия отражено в названии языка. Конечно, это понятие – **модуль**.

Как и в Аде, спецификация в Модуле-2 отделена от реализации. Представлены они соответственно определяющими (DEFINITION) и реализующими (IMPLEMENTATION) модулями, аналогами спецификации и тела пакета в Аде.

Продуманная интерпретация фундаментальной концепции модуля – основа элегантности и конкурентоспособности Модулы-2. Именно на этой интерпретации мы и сосредоточим свой анализ.

12.5.1. Управление сетями на Модуле-2

Ниже следует определяющий модуль ПараметрыСети (аналог спецификации соответствующего А-пакета).

```
1. DEFINITION  MODULE  ПараметрыСети;
2. EXPORT  QUALIFIED  МаксУзлов, МаксСвязей;
3.    CONST МаксУзлов = 100;
4.       МаксСвязей = 8;
5. END ПараметрыСети;
```

Как видите, очень похоже на Аду. Отличаются ключевые слова; в идентификаторах недопустимы разделители-подчеркивания (поэтому применяются большие буквы для отделения слов); допустимы серии объявлений типов, констант и переменных, выделяемых соответствующим ключевым словом; вместо is применяется знак «=». Короче говоря, Модула-2 в перечисленных отношениях ближе к своему старшему родственнику – Паскалю, чем Ада.

Главное отличие – во второй строке. Она представляет собой так называемый список экспорта. В нем явно перечисляются те и только те имена, определенные в модуле, которые считаются доступными (видимыми) в объемлющем контексте.

> Точнее говоря, ключевое слово QUALIFIED указывает на косвенный экспорт, когда доступны лишь полные имена (с указанием имени экспортирующего модуля): ПараметрыСети.МаксУзлов и ПараметрыСети.МаксСвязей. При отсутствии этого ключевого слова имеется в виду прямой экспорт – непосредственно доступны «короткие» имена МаксУзлов и МаксСвязей.

В использующих модулях можно управлять доступом с помощью так называемых списков импорта.

12.5.2. Определяющий модуль

```
1. DEFINITION MODULE УправлениеСетями;
2. FROM ПараметрыСети IMPORT МаксУзлов, МаксСвязей;
         (* это список импорта *)
3. EXPORT QUALIFIED Создать, Вставить, Удалить, Связать,
         Узел, Связи, Присвоить, УзелЕсть,
         ВсеСвязи, Сети;
         (* это список экспорта *)
4. TYPE  Узел = [1..МаксУзлов];
5.       ЧислоСвязей = [0..МаксСвязей];
```

```
6.        ИндексУзла = [1..МаксСвязей];
          (* производных типов нет. Все три типа совместимы.*)
7.        ПереченьСвязей = ARRAY ИндексУзла OF Узел;
          (* регулярный тип (тип массива). Все неформальные массивы — с постоянными
          границами. Точнее говоря, границы — константные выражения, вычислимые
          в период компиляции. *)
8.        Связи = RECORD
9.              Число :ЧислоСвязей;
                (* инициализации нет *)
10.             Узлы : ПереченьСвязей;
11.          END;
(* комбинированный тип (тип записи). Допустимы и вариантные.*)
12.       Сети;
(* указано только имя типа. Это так называемое непрозрачное объявление типа. Аналог
объявления приватного типа. *)
13.PROCEDURE Создать (VAR Сеть : Сети);
(* В Аде этого не было. Непрозрачные типы могут быть только ссылочными или отрезка-
ми предопределенных типов. Содержательные сети у нас — массивы. Поэтому тип "Сети"
будет ссылочным (а никак не отрезком). Реализовать процедуру создания
соответствующего массива можно только в модуле, экспортирующем тип "Сети", то есть
в реализующем модуле УправлениеСетями. Дело в том, что в отличие от Ады приватной
части в определяющем модуле нет. Поэтому нет во внешнем контексте и информации об
устройстве непрозрачных типов (ее нет даже для транслятора). Приходится определять
специальную процедуру создания содержательных сетей. *)
(* Параметр "Сеть" специфицирован ключевым словом "VAR" — это так называемый
параметр-переменная — аналог А-переменной, вызываемой в режиме in out. В результате
исполнения процедуры будет создан указатель на массив-сеть и присвоен формальному
параметру-переменной. *)
14.PROCEDURE Вставить(X : Узел; ВСеть : Сети);
(* Оба параметра — параметры-значения (аналог режима in). Хотя второй параметр
указывает на содержательно изменяющуюся сеть, сам указатель при этом остается
неизменным. *)
15.PROCEDURE Удалить(X : Узел; ИзСети : Сети);
16.PROCEDURE Связать(АУзел, ВУзел : Узел; ВСети : Сети);
17.PROCEDURE Присвоить(Сеть1, Сеть2 : Сети);
(* В Аде этого не было. Во внешнем контексте содержательное присваивание сетей
описать невозможно из-за отсутствия информации об их строении даже у транслятора —
приватной части нет! Поэтому и приходится определять специальную процедуру для
присваивания содержательных сетей. *)
18.PROCEDURE УзелЕсть(X : Узел; ВСети : Сети) : BOOLEAN;
   (* Так объявляют в Модуле-2 логическую функцию. *)
19.PROCEDURE ВсеСвязи(X : Узел; ВСети : Сети) : Связи;
20.END УправлениеСетями;
```

Строка 1 – заголовок определяющего модуля. Строка 2 – список импорта. Перечисленные в нем имена (экспортированные модулем ПараметрыСети) доступны (прямо, по коротким именам) в модуле УправлениеСетями. Важно, что никакие другие внешние имена (кроме стандартных) недоступны. Часть FROM списка импорта указывает имя модуля-экспортера и тем самым позволяет приме-

нять короткие имена. Если бы список начинался сразу словом IMPORT, то в нем должны были бы фигурировать косвенные имена (с точкой) для случая косвенного экспорта (и прямые имена для случая прямого экспорта).

Строка 3 – список косвенного экспорта (требующего во внешнем контексте в общем случае косвенных имен). Косвенный экспорт называют иногда «квалифицированным», а действие фрагмента FROM – «снятием квалификации». Такие обороты выгладят чужеродными в русском тексте.

В строке 12 – непрозрачное объявление типа. По назначению оно соответствует объявлению приватного типа в Аде. Во внешнем контексте становится известно имя объявленного непрозрачного типа, но применять к его объектам можно лишь фиксированный набор операций, объявленных в этом же (определяющем) модуле, так как о природе объектов непрозрачного типа во внешнем контексте ничего не известно.

Конечно, имя непрозрачного типа и имена соответствующих операций должны быть в списке экспорта. Иначе тип незачем объявлять непрозрачным.

12.5.3. Использующий модуль

```
MODULE ПостроениеСетей;
(* это главный модуль, ничего не экспортирующий. *)
(* определяющий модуль для главного не пишется. *)
      FROM УправлениеСетями IMPORT Создать, Вставить, Связать, Присвоить, Сети;
      VAR Сеть1, Сеть2 : Сети;
(* объявление переменных типа Сети. *)
BEGIN
Создать(Сеть1); (* содержательную сеть,*)
(* в отличие от объекта типа Сети – можно создать только*)
(* с помощью импортированной процедуры *)
   Создать(Сеть2);
   Вставить(33, 13, Сеть1);
   ...
   Присвоить(Сеть1, Сеть2);
(* объекту, указанному Сеть2, присваивается значение объекта, указанного Сеть1. См.
реализующий модуль, экспортирующий тип Сети. *)
END ПостроениеСетей;
```

В этом модуле отражено уже упоминавшееся важное ограничение, касающееся непрозрачных типов:

объектами непрозрачных типов могут быть только ссылки (указатели) или скалярные объекты.

Однако при использовании непрозрачных типов неизвестно, как они устроены. Поэтому с ними можно выполнять лишь операции, явно экспортированные соответствующим определяющим модулем. Именно поэтому по сравнению с Ада-реализацией добавились операции Создать и Присвоить.

Так что непрозрачные типы Модулы-2 по использованию близки к ограниченным приватным типам Ады.

12.5.4. Реализующий модуль

Ниже следует реализующий модуль (аналог тела пакета):

```
IMPLEMENTATION MODULE УправлениеСетями;
   TYPE ЗаписьОбУзле = RECORD
                     Включен : BOOLEAN;
                     Связан : Связи;
                     END;
        Сети = POINTER TO ARRAY Узел OF ЗаписьОбУзле;
(* описание устройства содержательных сетей. *)
(* действует правило последовательного определения. *)

   PROCEDURE Создать(VAR Сеть : Сети);
   BEGIN
   Сеть := NEW Сети;
(* Работает генератор динамического объекта. Создаются объект анонимного регулярно-
го типа (содержательная сеть) и указатель на этот объект. Созданный указатель
присваивается ссылочной переменной — параметру "Сеть". Обратите внимание, в генера-
торе Модула-2 используется ссылочный тип, а не базовый, как в Аде. Поэтому базовый
вполне может оставаться анонимным. *)
   END Создать;

   PROCEDURE УзелЕсть(X : Узел; ВСети : Сети) : BOOLEAN;
   BEGIN
      RETURN ВСети^[X].Включен;
(* "^" означает так называемое разыменование — переход от имени к его значению.
Явное разыменование в Модуле-2 применяется только для объектов ссылочных типов.
"ВСети^" означает массив, на который ссылается указатель "ВСети". Квадратные скобки
выделяют список индексов (алгольная традиция). Точка имеет тот же смысл, что и в
Аде. *)
   END УзелЕсть;

   PROCEDURE ВсеСвязи(X : Узел; ВСети : Сети) : Связи;
   BEGIN
         RETURN ВСети^[X].Связан;
   END ВсеСвязи;

   PROCEDURE Вставить(X : Узел; ВСеть : Сети);
   BEGIN
      ВСеть^|X].Включен:= TRUE;
      ВСеть^|X].Связан.Число := 0;
   END Вставить;

   PROCEDURE Присвоить(Сеть1, Сеть2 : Сети);
   BEGIN
      Сеть2^ := Сеть1^:
   END Присвоить;

   PROCEDURE Чистить(Связь, ВУзле : Узел; ВСети : Сети);
```

```
      VAR i : 1 ..МаксСвязей;
(* Переменную цикла нужно объявлять. При этом контроль диапазона менее жесткий, чем
в аналогичной А-программе, так как граница отрезка типа обязана быть константным
выражением. *)
   BEGIN
      FOR i:= 1 TO ВСети^[ВУзле].Связан.Число DO
         IF ВСети^[ВУзле].Связан.Узлы [i] = Связь THEN
            Переписать(ВУзле, i, ВСети);
         END (* условия *);
      END (*цикла *);
   END Чистить;
(* Мы сознательно программируем близко к соответствующей А-программе, хотя можно
было бы действовать рациональнее. *)

PROCEDURE Переписать (ВУзле : Узел;
                      После : ИндексУзла;
                      ВСети : Сети);
   VAR j : 1.. МаксСвязей;
BEGIN
      WITH ВСети^[ВУзле].Связан DO (* присоединяющий оператор *)
         FOR j := После TO Число-1 DO
            Узлы [j]:= Узлы [j+1];
         END (* цикла *);
         Число := Число-1;
      END (* присоединяющего оператора *)
END Переписать;
(* Вместо переименования (которого нет) с успехом применен так называемый присоеди-
няющий оператор вида
      WITH ИмяЗаписи DO Операторы END
Его смысл в том, что между DO и END селекторы полей записи, указанной посредством
ИмяЗаписи, доступны по коротким именам. В нашем случае это селекторы "Число" и
"Узлы". Присоединяющий оператор имеется и в Паскале. *)

PROCEDURE Удалить(X : Узел; ИзСети : Сети);
   VAR i : 1 ..МаксСвязей;
BEGIN
   ИзСети^[X].Включен := FALSE;
      FOR i:=1  TO ИзСети^[X].Связан.Число DO
         Чистить(X, ИзСети^[X].Связан.Узлы^[i], ИзСети);
      END (* цикла *);
END Удалить;

PROCEDURE Есть_связь(АУзел, ВУзел : Узел, ВСети : Сети): BOOLEAN;
      VAR i : 1..МаксСвязей;
BEGIN
         WITH ВСети(АУзел).Связан DO
            FOR i in 1..запись.число DO
               IF Узлы(i) = ВУзел THEN
                     RETURN TRUE;
```

```
            END;
        END;
        RETURN FALSE;
END Есть_связь;

PROCEDURE Установить_связь(Откуда, Куда : Узел; ВСети : Сети);
BEGIN
    WITH ВСети(Откуда).Связан DO
        Число:= Число + 1;
        Узлы (Число):= Куда;
END Установить связь;

PROCEDURE Связать(АУзел, ВУзел : Узел; ВСети : Сети);
BEGIN
        IF not Есть_связь(АУзел, ВУзел, ВСети) THEN
            Установить связь(АУзел, ВУзел, ВСети);
            IF АУзел /= ВУзел THEN
                Установить связь(ВУзел, АУзел);
            END;
        END;
END Связать;
END УправлениеСетями;
```

Подчеркнем, что во внешнем контексте доступны только имена из списка экспорта определяющего модуля. Реализующий модуль может включать лишь список импорта (когда в нем используются внешние имена, которые не потребовались в определяющем модуле). Как и в Аде, все имена, доступные в определяющем модуле, доступны и в его реализующем модуле.

Итак, поставленная задача полностью решена. Обеспечена аналогичная А-случаю целостность сетей, модифицируемость комплекса и надежность программирования.

Вопрос. За счет чего?

Ответ. Непрозрачный тип «Сети», модульность (в частности, разделение спецификации и реализации) и явные объявления (в частности, отрезки типов).

Основной вывод из нашего эксперимента: обычные программы можно писать на Модуле-2 практически с тем же успехом и комфортом, что и на Аде.

Конечно, такие выводы не делают на основании одного эксперимента с неотлаженным и тем более не применявшимся на практике комплексом программ. Однако для наших целей важно, что при решении поставленной задачи мы ни разу не попали в ситуацию, когда вместо А-средств не нашлись бы подходящие М-возможности, не приводящие к необходимости кардинально перерабатывать программу. При этом задача не подбиралась специально с учетом особенностей Модулы-2, а была просто взята первая задача, на которой мы изучали возможности Ады.

Накоплено достаточно сведений о Модуле-2, чтобы содержательно обсудить принцип чемоданчика в сопоставлении с принципом сундука. Однако так как при этом не обойтись без сопоставления А- и М-решений, уясним, в каком смысле

целесообразно их сопоставлять, насколько это может быть правомерно и почему поучительно.

Ключевые понятия при ответах на эти вопросы – **языковая ниша** и **авторская позиция**. О втором уже шла речь, а первым займемся в следующем разделе.

12.6. Языковая ниша

Вопрос о том, насколько правомерно сравнивать Модулу-2 с Адой как потенциальных конкурентов, тесно связан с понятием «языковая ниша».

Языковая ниша – это комплекс внешних условий, при которых активная жизнь двух различных языков невозможна. Языковая ниша (или просто ниша) характеризуется по меньшей мере классом пользователей ЯП, классом решаемых задач (проблемной областью), классом инструментальных и целевых компьютеров (точнее, программных сред, включающих компьютеры, операционные системы, используемые прикладные пакеты и т. п.). На нишу, соответствующую ЯП, влияют, кроме собственных свойств ЯП, также особенности технической и социальной поддержки (наличие и качество реализаций, экономическая или иная заинтересованность организаций, фирм, ведомств, стран и т. д.).

Сравнивать в качестве потенциальных конкурентов имеет смысл лишь ЯП, претендующие на одну и ту же или пересекающиеся ниши. Иначе различия между ЯП всегда можно объяснить различием «условий их активной жизни».

Однако если у конкретного ЯП сформировалась определенная ниша, то бесперспективно говорить о его вытеснении потенциальным конкурентом. Опыт показывает, что справедлив **закон консерватизма ниш**: ниша сопротивляется замене ЯП. Этот же закон можно понимать как закон сохранения (обитателей) ниш. Поэтому к ЯП (возможно, даже более, чем к программам) применим известный афоризм В. Л. Темова, который в применении к ЯП можно перефразировать так: «Языки не внедряются, а выживают».

Так что мало смысла обсуждать, например, замену Фортрана или ПЛ/1 на Аду или Модулу-2 без изменения класса используемых компьютеров, контингента пользователей, решаемых задач и (или) других характеристик ниши.

Однако сравнивать в качестве потенциальных конкурентов Аду с Модулой-2 имеет смысл, хотя в первом приближении это ЯП различного назначения. Ада, как уже говорилось, ориентирована в первую очередь на программирование встроенных (встраиваемых) систем с применением кросс-компиляции. А Модула-2 – на задачи системного программирования на относительно бедных однопроцессорных компьютерах.

Реальные языковые ниши для этих ЯП еще не сформировались. Они могут оказаться конкурентами в ситуациях, когда принцип чемоданчика проявит большую жизнеспособность, чем принцип сундука, даже подкрепленный мощной поддержкой директивных органов.

Так, еще до распространения доступных высококачественных реализаций Ады Модула-2 может «захватить» классы применений, где ресурсы целевого компьютера позволяют отказаться от кросс-компиляции, если сам компилятор достаточ-

но компактен. Подробное сопоставление потенциальных ниш Ады и Модулы-2 выходит за рамки книги.

Особенно поучительно сравнивать проектные решения с авторской позиции. В этом случае проектируемые ЯП могут быть предназначены для совершенно разных языковых ниш. Желательно лишь, чтобы сопоставляющий по возможности четко представлял себе особенности этих ниш и их связь с принимаемыми проектными решениями.

При таком подходе принцип чемоданчика как авторский принцип может проявиться в том, чтобы отказаться от борьбы за определенную нишу, если для этого не созрели технические или социальные условия. Другими словами, принцип минимальности можно интерпретировать и так, что следует выбирать ниши, где предлагаемые языковые решения будут выглядеть как совершенно необходимые.

Итак, будем сравнивать А- и М-решения прежде всего с авторской позиции. Однако учтем, что возможно и пересечение соответствующих ниш.

12.7. Принцип чемоданчика в проектных решениях ЯП Модула-2

12.7.1. Видимость

Занимаясь управлением видимостью в Аде, мы обнаружили, во-первых, технологические потребности, которые привели к созданию соответствующего А-аппарата (пакетов, блоков, указателей контекста и сокращений, правил перекрытия и наследования операций производных типов, переименования, неявных объявлений). Во-вторых, были продемонстрированы проблемы, вызываемые нежелательным взаимодействием многочисленных компонент этого сложного аппарата. В частности, было показано, как неявные объявления в сочетании с указателем сокращений могут приводить к неожиданным последствиям, явно входящим в противоречие с основной целью ЯП – обеспечить надежное программирование.

Как должен поступить автор ЯП, руководствующийся принципом чемоданчика? Он должен заново проанализировать потребности и поискать **компромисс**, в максимальной степени удовлетворяющий критические потребности при минимальной возможной сложности предлагаемого языкового аппарата. Ключевая идея такого компромисса, найденная Виртом для Модулы-2, – **полный отказ от любых косвенных (неявных) объявлений**.

Компромисс состоит в том, что в первом приближении такая идея должна быть не слишком удобной для пишущего программу, а иногда и для читающего. Ведь появляются (потенциально довольно длинные) списки экспортируемых и импортируемых имен. Их нужно писать, читать, проверять (и хранить). К тому же по ним все равно невозможно узнать свойства именуемых сущностей, и приходится как читателю, так и транслятору анализировать экспортирующие модули. Однако потому это и компромисс, что становятся совершенно тривиальными правила видимости и контроля имен – ведь все они объявлены явно в каждом модуле.

Вместе с тем этот компромисс следует именно принципу чемоданчика. Для обеспечения раздельной компиляции совершенно необходимо управлять видимостью имен, смысл которых определен в других модулях. Применяем простейший способ – явное перечисление нужных имен с указанием модуля-экспортера. Получаем списки импорта. Для надежности, облегчения понимания и модификации применяем двойственный конструкт – списки экспорта. Получаем логически завершенный аппарат связывания модулей.

У принятого решения – несколько важных следствий.

Во-первых, экспорт-импорт становится точнее, чем в Аде, где из-за правила последовательного определения видимыми могут оказаться несущественные для внешнего контекста, по сути промежуточные, имена.

Во-вторых, концепция явных объявлений несовместима с концепцией производных типов (с естественными для них неявными определениями операций). Весьма вероятно, что именно отмеченная несовместимость – одна из причин отсутствия производных типов в Модуле-2. А ведь это целый языковый пласт.

В-третьих, отказ от производных типов не позволяет применять типы в качестве характеристики содержательной роли объекта, что меняет саму концепцию типа. Такого рода последствия можно обнаруживать и дальше.

С одной стороны, они подчеркивают взаимозависимость и взаимообусловленность компонент ЯП как знаковой системы. С другой стороны, указывают цену компромисса. С третьей стороны, показывают, от каких заманчивых возможностей (например, контроль содержательных ролей) приходится отказываться ради простоты определения, реализации и использования ЯП, помня о неумолимом законе распространения сложности.

В связи с управлением видимостью интересно понять, почему в Модуле-2 остался присоединяющий оператор (остался от Паскаля). Казалось бы, это полный аналог указателя контекста и неявные (локальные) объявления полей записи – очевидное противоречие с концепцией явных объявлений.

Однако серьезного противоречия нет. Скорее, наоборот, можно и здесь усмотреть следование принципу чемоданчика. Только понимать его нужно глубже, трактуя «совершенно необходимо» не только в чисто техническом, но и в «социальном» смысле. Во-первых, область действия неявных объявлений полей строго ограничена – между DO и END одного оператора. Во-вторых, как было показано, становятся менее нужными переименования (с такими своими проблемами, как синонимия – доступ к одному объекту по различным именам; это очень ненадежно; **почему?**). В-третьих, присоединяющий оператор допускает весьма эффективную реализацию. Запись, с которой предполагается работать, можно разместить в сверхоперативной памяти, на рабочих регистрах и т. п. С этой точки зрения отказаться от присоединяющего оператора – решение сомнительное.

Однако, чтобы понять, почему присоединяющий оператор совершенно необходим в Модуле-2, нужно привлечь соображения социального характера, явно учитывающие потенциальную нишу для этого языка. Присоединяющий оператор совершенно необходим в Модуле-2 потому, что он имеется в Паскале (к нему привыкли те самые пользователи, которые с большой вероятностью начнут поль-

зоваться Модулой-2 как естественным преемником Паскаля (если угодно, его модульным диалектом)), то есть у этих ЯП – потенциально пересекающиеся ниши.

Обратите внимание, закон консерватизма ниш в данном случае работает не против нового ЯП, а за него, потому что Модулу-2 следует рассматривать не как конкурента Паскаля, а как его естественное развитие, учитывающее консерватизм ниши. **Правда, различные развития Паскаля вполне могут конкурировать (и реально конкурируют!) между собой и, в частности, с Модулой-2.**

12.7.2. Инкапсуляция

Особенно наглядно принцип чемоданчика проявляется в методе М-инкапсуляции. Обсуждая необходимость приватной части в Аде, мы привлекали реализаторскую позицию (соображения эффективности реализации: без приватной части компилятор не в состоянии распределять память под объекты приватных типов). И отмечали, что при этом нарушается согласованность с концепцией разделения спецификации и реализации, а также пошаговой детализации.

Другими словами, эффективность реализации в этом случае достигается за счет ряда нарушений общих принципов и усложнения языка (кстати, тоже нарушение общего принципа, а именно принципа чемоданчика).

Анализируем проблему по принципу чемоданчика. Что совершенно необходимо? Инкапсуляция как средство достижения надежности и целостности. Ищем компромисс между потребностями и возможностями простых проектных решений. Находим его в отказе от особо эффективной реализации (по сути – от статического распределения памяти под инкапсулированные объекты; сравните работу с А- и М-сетями). Следствие – возможность отказаться от весьма неприятной приватной части и тем самым обеспечить соблюдение четких принципов проектирования программы (точное разделение спецификации и реализации между двумя категориями модулей).

Главная цель достигнута. ЯП стал проще (и технологичнее), так как распределение памяти под составные инкапсулированные объекты должно выполняться не компилятором, а динамически – генератором «непрозрачных» указателей.

Ограничение непрозрачных типов ссылочными и отрезками предопределенных типов позволяет компилятору выделять память для них как для скаляров (по одной «единице» памяти). Остальное выполняется в динамике процедурами модуля-экспортера. Изящное решение.

> **Упражнение.** Докажите неформальную теорему: отсутствие приватной части в сочетании с раздельной компиляцией спецификаций и тел модулей влечет динамизм составных инкапсулированных типов. Как следствие – ограничение непрозрачных типов ссылочными или скалярными.

12.7.3. Обмен

Мы видели, как вся мощь А-модели использовалась для управления обменом. Но вся мощь потребовалась именно потому, что А-модель претендует на удовлетво-

рение отнюдь не минимальных потребностей. Например, можно создавать файлы с совершенно произвольными (а не только предопределенными) типами элементов – именно это потребовало, чтобы пакеты обмена стали родовыми. Аналогичные претензии удовлетворяются при создании драйверов специальных устройств обмена – именно это потребовало развитых спецификаций представления.

С другой стороны, для реализации драйверов привлекается универсальный аппарат управления асинхронными процессами. Когда он уже имеется в ЯП, такое решение может показаться даже изящным. Однако на уровне машинных команд организация взаимодействия с аппаратной задачей может существенно отличаться от организации взаимодействия «полностью программных» задач. Мы уже отмечали это, приводя пример драйвера клавиатуры.

Так что компилятор вынужден выделять драйверы и все-таки программировать их не так, как другие задачи. Выделять драйверы приходится по спецификациям представления. Итак, применение универсального аппарата асинхронных процессов для реализации драйверов заставляет сначала тщательно замаскировать то, что затем приходится столь же тщательно выискивать.

Создавать трудности, чтобы потом их преодолевать, – не лучший принцип не только в программировании.

Наконец, хотя асинхронность внешних устройств – одна из причин появления в ЯП асинхронных процессов, совершенно не очевидно, что для создания драйверов требуется столь абстрактный (и дорогой в реализации) аппарат, как рандеву из А-модели.

Итак, с учетом ориентации Модулы-2 в основном на однопроцессорные компьютеры (но с асинхронными устройствами обмена) взглянем на управление обменом, руководствуясь принципом чемоданчика.

Что совершенно необходимо? Дать возможность писать драйверы, обеспечивающие взаимодействие основной программы с асинхронно работающим устройством обмена.

Снова ищем **компромисс** между потребностями в асинхронных процессах и возможностями простых проектных решений.

Находим его в отказе, во-первых, от того, чтобы драйвер работал асинхронно с основной программой (оставаясь программной моделью аппаратной задачи), и, во-вторых, от абстрактного механизма взаимодействия относительно равноправных задач (подобного рандеву).

Действительно, **минимальные потребности** состоят в том, чтобы основная программа (точнее, ее часть, драйвер устройства) имела лишь возможности:

1) запустить устройство для выполнения конкретного обмена;
2) продолжать работать, пока устройство исполняет задание;
3) реагировать на завершение обмена (на факт выполнения устройством задания).

Именно такие минимальные (совершенно необходимые) возможности управления устройствами встроены в Модулу-2. Точнее говоря, то, что мы назвали аппаратной задачей, в Модуле-2 называется периферийным (асинхронным) процессом.

Периферией обычно называют совокупность устройств обмена. В Модуле-2 имеются еще и квазипараллельные процессы (сопрограммы).

Примеры периферийных процессов в Модуле-2

Рассмотрим на примере периферийные процессы в Модуле-2. Как и в Аде, в Модуле-2 обмен требует всей мощи ЯП. Существенно используется основной механизм абстракции – модули. Определяемые реализацией («системно зависимые») имена инкапсулированы в предопределенном модуле «Система» (SYSTEM). Так как транзит импортированных имен запрещен, то любые модули, где применяются системно зависимые имена, должны явно импортировать модуль «Система». (По этому признаку системно зависимые модули легко распознавать.)

Модуль «Система» экспортирует, в частности, типы «Адрес» (ADDRESS), «Слово» (WORD – машинное слово), «Процесс» (PROCESS – к этому типу относятся как сопрограммы, так и периферийные процессы), а также процедуры для работы с объектами этих типов: «НовыйПроцесс» (NEWPROCESS), «Переключить» (TRANSFER) и «ПереключитьСЗаказом» (IOTRANSFER).

Так что в системно зависимых модулях (в том числе и в драйверах) можно работать с машинными адресами, словами и процессами. Последние характеризуются двумя компонентами – телом (представленным некоторой процедурой) и рабочей областью, в свою очередь характеризуемой начальным адресом и размером (представленным натуральным числом).

Процедура

```
НовыйПроцесс(P, A, n, p1);
```

создает новый процесс (объект типа «Процесс» с телом P и рабочей областью с начальным адресом A и размером n) и присваивает его переменной p1 типа «Процесс». Новый процесс при этом не запускается (ведь процессор один), продолжает выполняться текущий процесс (основная программа также считается процессом).

Переключение на новый процесс осуществляется процедурой

```
Переключить(p1, p2);
```

При этом текущий процесс приостанавливается и присваивается переменной p1, а активным становится процесс-содержимое переменной p2. Напомним, что это сопрограммы; p2 начинает работать с начала своего тела или с того места, где ранее приостановился.

Процедура

```
ПереключитьСЗаказом(p1, p2, A);
```

делает то же, что и предыдущая, но еще и заказывает переключение снова на p1 после прерывания по адресу A.

Именно эта процедура и позволяет обеспечить указанные выше потребности 2) и 3). Для этого достаточно указать в качестве p2 процесс, который должен работать асинхронно (параллельно) с устройством, а в качестве адреса A указать адрес вектора прерываний, приписанный управляемому устройству.

Тогда если непосредственно перед выполнением процедуры Переключить-СЗаказом запустить обмен с устройством, то текущий процесс, приостановив-

шись на этой процедуре, будет ждать прерывания, свидетельствующего о завершении обмена. При этом параллельно с устройством будет работать процесс p2. А после прерывания произойдет заказанное переключение снова на p1, то есть на процесс, запустивший обмен с устройством (с заказом прерывания). Обычно обмен запускается засылкой единицы в соответствующий разряд регистра состояния устройства – так реализуется указанная выше потребность 1).

Вот такими скупыми средствами реализовано в Модуле-2 управление периферийными процессами. С точки зрения языка не понадобилось вообще ничего нового, а с точки зрения модуля «Система» – всего одна процедура ПереключитьСЗаказом. Так действует принцип чемоданчика! Чтобы лучше понять взаимодействие описанных средств, приведем (с переводом идентификаторов на русский язык) модуль обмена с телетайпом из авторского описания Модулы-2.

Драйвер на Модуле-2. Чтобы все в нижеследующей программе (модуле «Телетайп») было понятно, нужно сказать несколько слов о приоритетах процессов. Приоритет – это целое число, характеризующее срочность процесса. Приоритет связывается с каждым модулем и с каждым устройством, посылающим прерывания. Исполнение программы может быть прервано тогда и только тогда, когда приоритет прерывающего устройства выше приоритета исполняемого (текущего) процесса. Приоритет процессора (то есть приоритет текущего процесса) можно временно понизить процедурой УменьшитьПриоритет (LISTEN) из модуля «Система». Нужно это для того, чтобы разрешить прерывания от устройств.

```
1 MODULE Телетайп [4]; (* приоритет этого модуля равен 4 *)
2  FROM Система IMPORT Слово, Процесс, НовыйПроцесс,
Переключить, ПереключитьСЗаказом, УменьшитьПриоритет;
3  EXPORT Печатать;
4  CONST  N = 32; (* размер буфера литер *)
5  VAR n : INTEGER; (* текущее количество литер в буфере *)
6  Класть, Брать : [1..N]; (* индексы в буфере, отмечающие, куда класть и откуда
брать литеры *)
7  Буф : ARRAY [1..N] OF CHAR; (* буфер, массив литер *)
8  Дай, Возьми : Процесс;
9  РабОбл : ARRAY [0..20] OF Слово;
10 РегСост [177564B] : BITSET; (*регистр состояния телетайпа*)
11 РегБуф  [177566B] : CHAR; (* буферный регистр телетайпа *)

12 PROCEDURE Печатать (Лит : CHAR);
13 BEGIN
14    INC(n); (* предопределенная процедура; n := n + 1 *)
15    WHILE n > N DO УменьшитьПриоритет END;
16    Буф [Класть] := Лит;
17    Класть := (Класть MOD N) + 1; (* MOD — операция взятия по модулю;
                        Индекс "Класть" циклически пробегает буфер *)
18    If   n = 0 THEN
         Переключить(Дай, Возьми)
         END;
19  END Печатать;
```

```
20 PROCEDURE Драйвер;
21 BEGIN
22    LOOP
23       DEC (n); (* предопределенная процедура; n := n — 1; *)
24                   if   n < 0 THEN
                            Переключить(Возьми, Дай)
                            END;
25                   РегБуф:= Буф [Брать];
                            Брать := (Брать MOD N)+1;
26                   РегСост := {6};
(* шестой разряд инициирует обмен *)
27                   ПереключитьСЗаказом(Возьми, Дай, 64B);
28                   РегСост:= { }; (* обмен завершен *)
29       END;
30 END Драйвер;

31 BEGIN n:=0; Класть:=1; Брать:=1;
   (* Инициализация *)
32    НовыйПроцесс(Драйвер, ADR (РабОбл), SIZE (РабОбл), Возьми);
(* Предопределенные функции доставляют соответственно адрес и размер объекта *)
33    Переключить(Дай, Возьми);
34 END Телетайп;
```

Подробности о функционировании модуля Телетайп. Представим себе применение этого модуля по такой схеме:

```
35 MODULE Печать;
36    FROM Телетайп IMPORT Печатать;
37    CONST M = 100;
38    VAR Текст : ARRAY [1..N] OF CHAR;
   ...
39    FOR J:= 1 TO M DO
40       Печатать(Текст [J]);
41    END;
42 END Печать;
```

Проследим взаимодействие компонент программы, указывая обрабатываемые (выполняемые) номера строк.

Инициализация. В самом начале модуля Печать происходят связывание с модулем Телетайп и выполнение его «инициализирующих» строк 31–33. Создается процесс с телом Драйвер и присваивается переменной Возьми. С этого момента Возьми используется для идентификации сопрограммы, непосредственно работающей с внешним устройством.

> Ее принципиальное отличие от процедуры Драйвер состоит в том, что переключение на Возьми означает *продолжение* работы сопрограммы, а не вызов процедуры Драйвер (с ее начала).

Затем (строка 33) эта сопрограмма запускается и одновременно текущий процесс (то есть основная программа) присваивается переменной Дай и приостанавливается (перед выполнением строки 37).

С этого момента основная программа выступает как процесс Дай, а драйвер – как Возьми. **Названия оправданы тем, что основная программа подает литеры в буфер Буф, а драйвер забирает их оттуда.**

Итак, запомним, что строка 32 нужна для создания процесса Возьми, а строка 33 – для создания процесса Дай. Взаимодействие начинается.

Начало. Буфер пуст. После строки 33 управление достигает цикла 22 с условием n=0, свидетельствующим о пустом буфере. Поэтому после строки 23 в строке 24 следует переключение на основную программу Дай. **(Вернется оно в драйвер на строку 25!)** Так будет всегда, когда драйвер в своем основном цикле освобождает буфер и переключается при n=–1 на основную программу Дай.

Эта программа продолжается со строки 37, рано или поздно доходит до строки 40 и вызывает Печатать с очередной литерой текста. Через строку 14 при условии n=0 проходим на 16 и помещаем литеру в буфер. Строка 18 отправляет на драйвер (строка 25) при n=0 (несколько неестественном условии; ведь в буфере имеется одна литера).

Основное взаимодействие. Буфер не пуст и не полон. Извлекая очередную литеру из буфера (в строке 25), драйвер запускает обмен с внешним устройством в строке 26 (присваивая его регистру состояния 1 в шестом разряде и активизируя тем самым аппаратную задачу).

Принципиально важная для нас строка 27 приостанавливает драйвер, переключает управление на основную программу (в первый раз – на строку 19, то есть сразу же на 39) и заказывает прерывание по концу обмена очередной литеры. Это прерывание (от телетайпа) в соответствии с семантикой процедуры ПереключитьСЗаказом приводит к переключению от Дай снова на Возьми в момент окончания обмена.

Пока идет обмен (работает аппаратная задача асинхронно с исполнением процессов Дай и Возьми), процесс Дай в цикле 39–41 может наполнять буфер. После прерывания драйвер в цикле 22–29 очищает буфер по одной литере. Это и есть основное взаимодействие процессов Дай и Возьми. При этом скорости заполнения и очистки буфера жестко не связаны.

> **Вопрос.** За счет чего буфер может очищаться быстрее, чем наполняться, и наоборот?

Особые ситуации. Буфер полон и пуст. Основное взаимодействие прекращается, если буфер оказывается полным (в строке 15 $n > N$) или пустым (в строке 24 $n < 0$).

Когда буфер полон, необходимо дать приоритет процессу Возьми, очищающему буфер, приостановив заполняющий процесс Дай. Это реализует цикл уменьшения приоритета (строка 16). Ведь по логике модуля Телетайп заполнение буфера более чем на одну позицию возможно только одновременно с работой аппаратной задачи (собственно обменом или ожиданием ею разрешения на прерывание (убедитесь в этом!)).

Поэтому переполнение буфера означает, что нужно обеспечить беспрепятственное выполнение очищающего цикла драйвера. Для этого процесс Дай и за-

держивается на цикле 15, в конечном итоге уступая (единственный!) процессор драйверу (при достаточном понижении приоритета). И буфер начинает очищаться.

Когда же буфер пуст, то строка 24 переключает управление на Дай с n=–1. Это соответствует уже разобранной ситуации «Начало. Буфер пуст».

Еще одно решение. Не видно причин, почему не написать модуль Телетайп концептуально проще, изъяв строку 33 и (как следствие) попадание в зону отрицательных n (такие значения не соответствуют назначению этой переменной – считать количество литер в буфере).

Упражнение. Найдите это решение.

(Пишем только «Печатать» и «Драйвер» при условии, что строки 33 нет.)

```
PROCEDURE Печатать(Лит : CHAR);
BEGIN
   WHILE n = N DO УменьшитьПриоритет END;
   Буф [Класть]:= Лит; INC(n); Класть:= (Класть MOD N) + 1;
   If  n=1 THEN Переключить(Дай, Возьми) END;
END Печатать;

PROCEDURE Драйвер;
BEGIN
      LOOP
         РегБуф := Буф [Брать]; DEC(n); Брать := (Брать MOD N) +1;
         РегСост:={6}; ПереключитьСЗаказом(Возьми, Дай, 64B);
         РегСост := {};
         If n=0 THEN Переключить(Возьми, Дай) END;
      END (* цикла *);
END Драйвер;
```

Упражнение. Докажите эквивалентность первому решению.

12.8. Принцип чайника

Обсуждая методы борьбы со сложностью программирования, полезно обратить внимание на принцип, интуитивно хорошо знакомый опытным программистам и выражающий своего рода защитную реакцию на сложность и ненадежность операционной среды.

Суть этого принципа хорошо иллюстрирует старый анекдот:

«Как вскипятить чайник?

– Берем чайник, наливаем воду, ставим на огонь, доводим до кипения.

Как вскипятить чайник, уже наполненный водой?

– Выливаем воду из чайника и сводим задачу к предыдущей!»

Почему математики так любят «сводить задачу к известной»? Потому, что для них главное – ясность («прозрачность», «надежность») доказательства, а прямое решение новой задачи рискует оказаться ошибочным.

Но ведь и для программистов главное – надежность и понятность программы. Поэтому опытный программист без особой нужды не станет пользоваться элементами операционной среды, которые он лично не проверил.

Это относится, в частности, к использованию отдельных команд языковых конструктов, программ, пакетов, а также ЯП. Важными оказываются не столько их свойства сами по себе, сколько то, что программист эти свойства знает и этому своему знанию доверяет.

Если окажется возможным «свести задачу к предыдущей», она будет, как правило, решена традиционными, обкатанными методами. Намекая на упомянутый анекдот, назовем соответствующий технологический принцип «**принципом чайника**».

Очевидное проявление принципа чайника – долгожительство классических ЯП, в особенности Фортрана. Менее очевидное (указанное впервые Дональдом Кнутом и затем многократно подтвержденное другими исследователями) – пристрастие программистов к самым тривиальным оборотам (фразам) при использовании ЯП.

Если шаг цикла, то 1; если прибавить, то 1; если присвоить, то простейшее выражение; если проверить, то простейшее отношение и т. п.

Принцип чайника помогает обосновать принцип чемоданчика (на этот раз уже с точки зрения психологии пользователя) – необязательными, чересчур изощренными конструктами будут редко пользоваться, они окажутся экономически не оправданными.

12.9. ЯП Оберон

В феврале 1988 г. стало известно о новом языке Н. Вирта – ЯП Оберон. Вирт не склонен связывать с выбором имени для своего очередного детища каких-либо глубоких соображений, однако отмечает, что для него «Оберон» – скорее самый крупный спутник Урана, чем король эльфов. Для нас Оберон интересен прежде всего как очередная попытка достичь идеала ЯП, следуя принципу чемоданчика с учетом новейших достижений в философии и технологии программирования.

Не вдаваясь в подробный анализ проектных решений, отметим лишь, что целью Вирта был минимальный базовый ЯП для персональной рабочей станции.

Более того, правильнее назвать требуемый ЯП не просто базовым, а монопольным (интегрированным) ЯП [20]. Другими словами, ЯП должен быть таким, чтобы ни создателю программного обеспечения станции (включая ее операционную систему), ни пользователю станции просто не нужен был никакой иной инструмент программирования. Конечно, минимальное ядро реализации любого ЯП должно быть написано на ассемблере. Но этим и должно ограничиваться применение иного ЯП.

Идея монопольного ЯП, с одной стороны, очевидным образом перекликается с идеей единого универсального ЯП, которая, как известно, многократно терпела фиаско и вновь воскресала на очередном этапе развития программирования. Ясно, что идея монопольного ЯП жизнеспособнее за счет отказа от претензий на

пригодность для любых классов задач, классов пользователей, любых компьютеров и программных сред. Более того, она фактически реализована в таких ЯП, как Си для UNIX-совместимых сред, Эль-76 для отечественной серии «Эльбрус», Том в одноименной интегрированной системе В. Л. Темова и др. Еще раз подчеркнем, что идеальный монопольный ЯП должен быть не просто принципиально возможным, а реально наилучшим инструментом программирования в своей среде. Тем более интересно посмотреть, как справляется с задачей создания минимального монопольного ЯП такой всемирно признанный мастер, как Вирт.

12.9.1. От Модулы-2 к Оберону

Укажем отличия Оберона от Модулы-2, следуя [21].

Главная новинка – средства обогащения (extension) комбинированных типов данных. Этот новейший аспект в ЯП мы подробнее рассмотрим в разделе, посвященном наследованию. Основная идея обогащения связана с воплощением «древней» мечты программистов – вводить при необходимости дополнительные поля в записи таким образом, чтобы сохранялась работоспособность всех ранее отлаженных программ. Один из известных учебников по структурному программированию [7] начинается с притчи о злоключениях программистов, не предусмотревших вовремя нужного поля. Вирт снимает все такого рода проблемы, предоставляя возможность обогащать комбинированный тип новыми полями с наследованием всех видимых операций исходного типа.

Например, если задан тип

```
T = RECORD x. y: INTEGER END,
```

то можно определить обогащенные типы

```
T1= RECORD(T) z: REAL END,
T2 = RECORD(T) w: LONGREAL END,
```

наследующие все видимые операции, определенные для T. При этом «обогащенные» объекты типов T1 и T2 можно присваивать «бедным» объектам типа T, а «бедные» «богатым» – нельзя. Обратите внимание, что не только в Модуле-2, но и в Аде подобное невозможно.

Вопрос. Почему такое «странное» правило присваивания? Ведь в обогащенную запись несложно разместить записи с меньшим числом полей, но это как раз запрещено, в то время как обратное разрешено, хотя вся запись наверняка не поместится.

Упражнение. Попытайтесь уточнить правило присваивания, предложив вариант размещения обогащенной записи в бедной.

Подсказка. Основной критерий – надежность программирования и применимость старых операций к новым объектам.

Конечно, подобное нововведение требует иногда пожертвовать эффективностью программы ради удобства ее изготовления, надежности и других преимуществ. Но в этом и состоит истинный прогресс в ЯП – **осознается фундаментальное значение компромиссов, казавшихся ранее немыслимыми**.

Вопрос. За счет чего может снижаться эффективность программы?

Подсказка. Не обойтись без указателей там, где ранее обходились.

Еще одно важное нововведение (точнее, коррекция исходного понятия) – трактовка спецификации как усеченной (без каких-либо добавлений!) реализации. Другими словами, спецификация полностью состоит из цитат, взятых из текста реализации, причем вне модуля видимо то и только то из реализации, что «проявлено» в спецификации. Назовем это идеей «**экспортного окна**», чтобы подчеркнуть, что экспортируется не более того, что имеется в реализации.

Вопрос. Что это дает?

Подсказка. Смотрите перечень средств из Модулы-2, не вошедших в Оберон.

Вопрос. Знаете ли вы примеры ЯП, где в спецификации может оказаться не только содержимое реализации?

Основные сокращения по сравнению с Модулой-2. Сокращены в основном средства, которые функционально перекрываются обогащением типов, а также некоторые средства, по мнению Вирта, не оправдавшие себя в базовом ЯП.

Типы данных:

- нет вариантных комбинированных типов – вместо них работают обогащенные;
- нет закрытых (непрозрачных) типов – вместо них работает общая идея управляемого «проявления» компонент реализации за счет их цитирования в спецификации. Особенно красиво это взаимодействует с обогащением типов. Например, если в реализации (теле модуля) содержатся объявления приведенных выше типов T, T1, T2, то в спецификации можно указать

  ```
  TYPE T1 = RECORD z: REAL; END;
  ```

  скрыв не только некоторые поля, но и «происхождение» типа. Вместе с тем, процитировав спецификации нужных операций, легко сделать их (и только их) доступными пользователям типа T1. Все содержательные возможности закрытых и приватных типов при этом сохранены (так как в Обероне спецификации и реализации размещаются и транслируются обязательно вместе);
- нет перечисляемых, поддиапазонов, тип множества только один (предопределенный над целыми), нет типа CARDINAL.

Вопросы. Как именно работают обогащенные типы вместо вариантных комбинированных? О каких возможностях приватных типов идет речь?

Другие аспекты:

- упрощен экспорт-импорт, ликвидирован предопределенный модуль SYSTEM с предопределенными типами ADRESS и WORD;
- убраны все средства для управления асинхронным исполнением процессов;
- убран оператор цикла (FOR);
- убран оператор присоединения (WITH) (точнее, он сильно переделан – превратился в оператор для защиты правильности обращения с обогащенными типами);

- убрано даже понятие программы – пользователь видит на экране меню, составленное из спецификаций нужных ему модулей, – это и есть перечень предоставляемых ему услуг. В самом начале – предопределенное меню.

12.9.2. Управление сетями на Обероне

Приведем переписанный на Обероне пример УправлениеСетями с краткими комментариями, подчеркивающими отличия Оберона от Модулы-2. Для удобства сопоставления сохранены старые номера строчек (отсутствие номера означает, что соответствующая строчка убрана совершенно). Так как в Обероне нет поддиапазонов, структура объявлений типа упрощена, но эквивалентных проверок мы в программу не вставляем для простоты. Ясно, что надежность страдает. Вирт, по-видимому, руководствовался таким принципом: **затраты должны быть видимы программисту** (должны требовать и его усилий – записывая явные проверки, программист лучше чувствует их стоимость, чем в случае автоматических проверок, вставляемых компилятором). Подобные соображения (с учетом упрощения транслятора) и привели к удалению поддиапазонов из Оберона.

```
1. DEFINITION ПараметрыСети;
(* это заголовок спецификации (сопряжения) модуля *)
3. CONST МаксУзлов = 100;
4.       МаксСвязей = 8;
5. END ПараметрыСети;
(* Списка экспорта в Обероне нет. Клиентам модуля ПараметрыСети доступны все имена,
объявленные в его спецификации *)

1. DEFINITION УправлениеСетями;
2. IMPORT  П: ПараметрыСети;
(* МаксУзлов, МаксСвязей убраны; в списке импорта — только имена модулей; "П" —
локальное имя; список экспорта не нужен *)
4. TYPE
         Узел = SHORTINT; (* встроенный тип *)
7.    ПереченьСвязей = ARRAY П.МаксСвязей OF Узел;
(* индексы всегда целые, нижняя граница — 0, верхняя — МаксСвязей-1 *)
8.    Связи = RECORD
9.        Число : SHORTINT;
10.       Узлы : ПереченьСвязей;
11.   END;
12. Сети = RECORD END;
(* Аналог объявления приватного (закрытого, непрозрачного) типа *).

(* 13. PROCEDURE Создать(VAR Сеть : Сети);
Как и в Аде, снова необязательна процедура динамического создания сетей — закрытые
типы реализуются тем же аппаратом, что и обогащаемые, — за счет встроенных указате-
лей; объекты таких типов могут быть и статическими. *)

14. PROCEDURE Вставить(X : Узел;
                       VAR ВСеть : Сети);
```

```
(* Обратите внимание на режим второго параметра! *)
15. PROCEDURE Удалить(X : Узел; VAR ИзСети : Сети);
16. PROCEDURE Связать(АУзел, ВУзел : Узел; VAR ВСети : Сети);
17. PROCEDURE Присвоить(VAR Сеть1,
                       Сеть2 : Сети);
(* В Аде последней процедуры не было. Как и в Модуле-2, во внешнем контексте
операцию содержательного присваивания сетей описать невозможно из-за отсутствия
информации об их строении. Присваивать (полные) значения объектам таких типов,
которые объявлены в видимой части модуля, в Обероне нельзя. Так что тип Сети —
аналог ограниченных приватных Ады. Поэтому приходится определять специальную
процедуру для присваивания содержательных сетей. *)

18. PROCEDURE Узел Есть(X : Узел;
                   VAR ВСети : Сети) : BOOLEAN;
19. PROCEDURE ВсеСвязи(X : Узел;
             VAR ВСети : Сети; VAR R : Связи);
(* В Обероне, как и в Паскале, результат функции — только скаляр. *)
20.END УправлениеСетями;


DEFINITION Клиент; (* программы — главного модуля в Обероне нет! *)
IMPORT У: УправлениеСетями;
PROCEDURE ПостроениеСетей;
END Клиент;

MODULE Клиент;
IMPORT У: УправлениеСетями;
PROCEDURE ПостроениеСетей;
      VAR Сеть1, Сеть2 : У.Сети; (* объявление переменных типа Сети *)
BEGIN
   У.Вставитъ(33, 13, Сеть1);
   У.Присвоить(Сеть1, Сеть2);
END ПостроениеСетей;
END Клиент;
```

Вопрос. Как же воспользоваться таким модулем?

```
MODULE УправлениеСетями;
   IMPORT П: ПараметрыСети;
   TYPE
      Узел = SHORTINT;
      ПереченьСвязей = ARRAY П.МаксСвязей OF Узел;
      Связи = RECORD
                  Число : SHORTINT;
                  Узлы : ПереченьСвязей;
               END;
      ЗаписьОбУзле = RECORD
                        Включен : BOOLEAN;
                        Связан : Связи;
                        END;
```

```
      Сети = RECORD C: ARRAY П.МаксУзлов OF ЗаписьОбУзле
END;
(* Приходится так определять тип Сети, чтобы можно было скрыть поле C; другого
способа строить закрытый тип в Обероне нет! *)

   PROCEDURE УзелЕсть(X : Узел; VAR ВСети : Сети) : BOOLEAN;
   BEGIN
      RETURN ВСети.C[X].Включен; (* вместо указателя — поле C *)
   END УзелЕсть;
```

Вопрос. Зачем второй параметр получил режим VAR?

Подсказка. Нужно ли копировать сеть?

```
PROCEDURE ВсеСвязи(X : Узел; ВСети : Сети;
                 VAR R : Связи);
BEGIN
   R := ВСети.C[X].Связан;
END ВсеСвязи;


PROCEDURE Вставить(X : Узел;
                 VAR ВСеть : Сети);
BEGIN
   ВСеть.C[X].Включен := TRUE;
   ВСеть.C[X].Связан.Число := 0;
END Вставить;

PROCEDURE Присвоить(VAR Сеть1,
                   Сеть2 : Сети);
BEGIN
   Сеть2.C := Сеть1.C; (* вне модуля такого не сделаешь *)
END Присвоить;

PROCEDURE Есть_связь(АУзел, ВУзел : Узел,
            VAR ВСети : Сети): BOOLEAN;
   VAR i : 1..П.МаксСвязей;
         z : Связи;
BEGIN
   z := ВСети.C[АУзел].Связан;
(* вместо присоединяющего оператора *)
   i:=0;
   REPEAT (* цикла FOR в Обероне нет *)
   IF z.Узлы(i)= ВУзел THEN
      RETURN TRUE;
   END;
   i :=i + 1;
   UNTIL i < z.Число
   RETURN FALSE;
END Есть_связь;
```

```
PROCEDURE Установить_связь(Откуда,
            Куда : Узел; VAR ВСети : Сети);
   VAR z: Связи;
BEGIN
   z:= ВСети.С[АУзел].Связан;
(* вместо присоединяющего оператора *)
   z.Число :=z.Число+1;
   z.Узлы(z.Число):= Куда;
END Установить связь;
PROCEDURE Связать(АУзел, ВУзел : Узел;
                 VAR ВСети : Сети);
BEGIN
   IF ~ Есть_связь(АУзел, ВУзел, ВСети) THEN
   (* «~» — отрицание *)
      Установить_связь(АУзел, ВУзел, ВСети);
      IF АУзел #ВУзел THEN
      (* «#» в Обероне — знак неравенства *)
         Установить_связь(ВУзел, АУзел);
      END;
   END;
END Связать;

PROCEDURE Переписать(ВУзле : Узел;
                    После : SHORTINT;
                    VAR ВСети : Сети);
   VAR j : SHORTINT;
BEGIN
(* присоединяющий оператор в Обероне отсутствует *)
   j:= После;
   WHILE j > ВСети.С[ВУзле].Связан.Число-1 DO
      ВСети.С[ВУзле].Связан.Узлы [j] :=ВСети.С[ВУзле].Связан.Узлы [j+1];
      j:= j+i;
   END
END Переписать;

PROCEDURE Чистить(Связь, ВУзле : Узел;
                 VAR ВСети : Сети);
VAR i : SHORTINT;
BEGIN
   i:=0;
   REPEAT
      IF ВСети.С[ВУзле].Связан.Узлы [i]= Связь THEN
         Переписать (ВУзле, i, ВСети);
         ВСети.С[ВУзле].Связан.Число := ВСети.С[ВУзле].Связан.Число-1;
         EXIT;
      END; i := i+1;
   UNTIL i < ВСети.С[ВУзле].Связан.Число
END Чистить;
```

```
(* Мы сознательно программируем близко к А- и М-программам, хотя можно было бы
действовать рациональнее. Видно, что отсутствие присоединяющего оператора мешает —
мы использовали два варианта его замены. *)

   PROCEDURE Удалить(X : Узел;
                     VAR ИзСети : Сети);
      VAR i : SHORTINT;
   BEGIN
      ИзСети.С[X].Включен := FALSE; i := 0;
      REPEAT
         Чистить(X, ИзСети.С[X].Связан.Узлы[i], ИзСети);
         i:= i+1;
      UNTIL i < ИзСети.С[X].Связан.Число
   END Удалить;
END УправлениеСетями;
```

Итак, задача полностью решена. Обеспечена аналогичная А- и М-случаям целостность сетей и модифицируемость. Надежность программирования несколько пострадала.

Вопрос. Чем это может повредить при управлении сетями?

Подсказка. Не всякий недостаток ЯП должен сказываться на любой программе.

Как и в случае с Модулой-2, можно заключить, что обычные программы можно писать на Обероне почти с тем же успехом и комфортом, что на Аде или на Модуле-2. Вместе с тем удалось почувствовать и неудобства от ликвидации присоединяющего оператора и привычных циклов.

Закончим краткое знакомство с ЯП Оберон утверждением, что при всей своей «аскетичности» он вполне пригоден для выполнения роли монопольного языка персональной рабочей станции в основном за счет двух мощнейших средств – высокоразвитой модульности, опирающейся на идею экспортного окна и обогащаемых типов. Работа первого из них показана, а вторым займемся в разделе о наследуемости.

Интересно (и поучительно) отметить, что оба этих средства суть два взаимно дополнительных (дуальных) проявления одного и того же известнейшего математического понятия, оказавшегося, как недавно выяснилось, полезным для понимания «момента истины» в самых современных концепциях программирования. Такое понимание позволяет отделять «зерна от плевел», принимать решения при развитии и стандартизации ЯП. Подробнее об этом сказано в разделе о наследуемости в ЯП.

Упражнение (повышенной трудности). Попытайтесь самостоятельно догадаться, о каком математическом понятии идет речь.

Часть II

Перспективы языков программирования

Глава 1. Перспективные модели языка 269
Глава 2. Функциональное программирование (модель Б) 289
Глава 3. Доказательное программирование (модель Д) 315
Глава 4. Реляционное программирование (модель Р) 339
Глава 5. Параллельное программирование в Оккаме-2 (модель О) 353
Глава 6. Наследуемость (к идеалу развития и защиты в ЯП) 391
Глава 7. Объектно-ориентированное программирование 415
Глава 8. Заключительные замечания 439

Глава 1

Перспективные модели языка

1.1. Введение 270
1.2. Операционное программирование – модель фон Неймана (модель Н) 271
1.3. Ситуационное программирование – модель Маркова-Турчина (модель МТ) 273

1.1. Введение

Если в первой части книги мы стремились дать представление по возможности о всех аспектах современного языка индустриального программирования, то во второй части наша главная цель – дать представление о перспективах и тенденциях развития ЯП. Конечно, и в первой части нас интересовали прежде всего понятия, принципы и концепции фундаментального характера, которые могут претендовать на долгую жизнь в области ЯП. Мы особенно подчеркивали ситуации, когда такой подход позволял прогнозировать развитие ЯП.

Вместе с тем в целом мы сознательно ограничили себя рамками одного стиля программирования, часто называемого операционным (операторным, фон-неймановским, традиционным, классическим и т. п.), представителями которого выступают практически все упоминавшиеся нами ЯП. Такая ограниченность была оправдана, пока нас интересовали по возможности все аспекты практического программирования в их взаимно согласованном воплощении в целостной знаковой системе. Именно поэтому был выбран и единый язык примеров – Ада, а все сопоставления обычно делались с ЯП аналогичного стиля, тем более что операционный стиль явно доминирует в ЯП массового программирования.

Однако рамки одного стиля становятся тесными, если нас интересуют тенденции и перспективы развития ЯП. Маловероятно, что в ближайшей перспективе какой-либо иной стиль программирования вытеснит операционный. Отсутствует и пример современного; ЯП, который вобрал бы в себя практически все накопленное богатство в этой области. Однако в ЯП практического программирования попадает лишь то, что предварительно проверено в теории и эксперименте. Поэтому знать иные стили и подходы полезно каждому, кто желает понимать, чего можно ждать от будущих ЯП. Наконец, знакомство с нетрадиционным подходом и оригинальным взглядом на казалось бы хорошо знакомые сущности доставляет ни с чем не сравнимое удовольствие.

За некоторыми стилями программирования закрепились вполне определенные названия, другие общепринятых названий не имеют. Мы позволим себе употреблять те названия, которые, по нашему мнению, в достаточной степени отражают суть рассматриваемого подхода. Вместо слов, например, «операционный стиль программирования» иногда говорят короче: «операционное программирование». Будем поступать аналогично по отношению ко всем стилям.

Рассмотрим несколько моделей ЯП, представляющих операционное, ситуационное, функциональное, доказательное, реляционное, параллельное и объектно-ориентированное программирование.

Первая из них играет чисто историческую роль «начала координат». Вместе с тем она предоставляет возможность на содержательно хорошо знакомом материале познакомить с весьма общими понятиями, характерными для математической позиции. В дальнейшем эти понятия применяются к менее знакомому материалу при рассмотрении других моделей.

Остальные модели непосредственно предназначены для представления определенных перспективных тенденций в области ЯП и программирования в целом.

1.2. Операционное программирование – модель фон Неймана (модель Н)

Рассмотрим модель, отражающую свойства первых ЭВМ, – модель весьма примитивную, но способную послужить для нас точкой отсчета. Опишем ее в соответствии с концептуальной схемой на стр. 40.

Базис. Два скалярных типа данных: адреса и значения. Конечный набор базисных скалярных операций (система команд): присваивание, условные операции, останов и др. Единственная структура данных – кортеж ячеек (то есть пар адрес -> значение) с линейно упорядоченными адресами (память). Есть выделенная ячейка С (регистр команд), в которой хранится адрес подлежащей выполнению команды. Никакой явной структуры операций, каждая операция сама определяет своего преемника (в том смысле, что модифицирует содержимое регистра команд).

Развитие. Никаких выделенных явно средств развития – все они скрыты в универсальности набора операций, среди которых ключевую роль играет оператор присваивания ячейке нового значения и зависимость выбора преемника от состояния памяти. Любое развитие возможно только путем явного моделирования новых операций за счет универсальности системы команд. Что такое значение, не уточняем. Достаточно считать, что это целые и строки литер (для выделенных ячеек ввода-вывода).

Защита. Полностью отсутствует.

Исполнитель. Память – базисная структура данных (кортеж ячеек), процессор – устройство, последовательно выполняющее указанные в С операции, поведение – последовательность состояний памяти, план (программа) – исходное состояние (или его выделенная часть), результат – заключительное состояние (если оно есть; при этом содержательно результатом обычно служит лишь выделенная часть заключительного состояния).

Указанные «части» для каждой программы свои. Так что в общем случае программа формально не отличается от исходных данных и результатов – одни и те же ячейки исполнитель может интерпретировать либо как содержащие команды, либо как содержащие данные. Все дело в том, в какой роли адреса ячеек используются в исполняемых командах.

Знаки и денотаты в модели Н. Сведения о базисе можно выразить с помощью следующих обозначений.

Пусть A – тип данных «адрес» (то есть множество адресов в модели Н), V – тип данных «значение» (то есть множество содержимых ячеек с адресами из A). Тогда конкретное состояние памяти можно представить функцией s типа

S: A->V,

то есть конкретным отображением адресов в значения. Обратите внимание, мы применяем метаязык (то есть язык для описания языков), в котором допустимы функциональные типы, причем структура функционального типа S описана парой

A->V,

где A – область определения (область отправления) функций этого типа, а V – область значений (область прибытия) функций этого типа.

Итак, состояние памяти (содержательно это кортеж содержимых ячеек) представлено формально функцией из адресов в значения (хранящиеся по этим адресам).

Тип функции «состояние» выражает **первый принцип фон Неймана – принцип произвольного доступа к памяти** (в конкретном состоянии s из S равнодоступны все ячейки; задай адрес – получишь значение).

Операции (операторы) в модели H – это объекты типа

St: S->S.

Кроме того, модель фон-Неймана характеризуется функцией декодирования операций (частично определенной)

d: V->Com,

где Com – команды, то есть операции, встроенные (элементарные, предопределенные) в H.

Второй принцип фон Неймана – **принцип хранимой программы** – отражается формулой

(∀ c из Com) (∃ v из V):d(v) = c,

где ∀ обозначает «для всех», а ∃ – «существует», то есть всякую команду можно записать в память (найдется способ ее закодировать).

Фактически здесь использованы элементы некоторого языка для описания семантики ЯП – семантического метаязыка. Язык для описания синтаксиса ЯП знаком из курса программирования. Таким синтаксическим метаязыком служит, например, БНФ (форма Бэкуса-Наура).

***Основное семантическое соотношение в модели H* (денотационная семантика).** Каков денотат программы s в модели H? Другими словами, какова та функция, которую реализует программа s?

Рассмотрим функцию r типа St

r: S->S,

которая обозначает результат выполнения программы s, то есть r(s) – это состояние s1, в котором выполняется операция остановки (stop). Оно не всегда достигается, то есть функция r – частично определенная, ведь не всякое состояние может служить программой и не всякая программа завершает работу.

Так что если **C** – регистр команд, то

d(s1(s1(C))) = stop

(такому семантическому соотношению удовлетворяет заключительное состояние si).

Обозначим через k=d*s*s композицию функций d,s,s. Тогда основное семантическое соотношение, определяющее денотат r(s) программы s в модели H, записывается так:

r(s) = (если k(C)=stop, то s, иначе r(k(C)(s))).

Другими словами, нужно выполнить над состоянием s операцию, получающуюся декодированием содержимого ячейки с адресом, взятым из **C**, и вычислить функцию r от полученного нового состояния, пока не окажется, что нужно выполнить операцию stop.

Что можно извлечь из формулы для r?

Во-первых, то, что один шаг выполнения программы требует в общем случае трех обращений к памяти (в нашей модели регистр команд – в основной памяти), ведь переход к новому состоянию описывается как d(s(s(C)))(s).

Во-вторых, становится еще очевиднее, что средства развития в модели Н не выделены – денотат программы разлагается лишь на очень мелкие части – денотаты отдельных команд (выраженные, кстати, функцией k). Отсутствуют средства для явного обозначения композиций функции k, то есть для явного укрупнения денотатов.

Функциональная (денотационная) семантика. Пусть P = {p} – множество программ, R={r} – множество функций типа S->S. Функциональной, или денотационной, семантикой программ называют функцию типа P->R, отображающую программу (то есть исходное состояние p) в соответствующую ей функцию r, удовлетворяющую основному семантическому соотношению.

Название «денотационная» возникло исторически. Всякая семантика денотационная в том смысле, что сопоставляет знаку (программе) некоторый ее денотат (смысл).

Обратите внимание, насколько сложной концептуально оказалась знаковая система Н. Во всяком случае, нам потребовались функции высших порядков (то есть функции, среди аргументов и (или) результатов которых встречаются снова функции). Функции такого рода математики часто называют «операторами».

Действительно, посмотрите на перечень примененных функций:

s: A --> V;
St: S --> S;
d: V --> Com;
r: S --> S;
sem: P --> R.

Очевидно, что и операция, отображающая состояния (функции из адресов в значения), и декодирующая функция – функции высшего порядка, как и семантическая функция по отношению к функциям p и r (первая из них отображает адреса в значения, вторая – исходное состояние в заключительное).

Примеры программ в модели Н можно найти в любом учебнике по программированию.

1.3. Ситуационное программирование – модель Маркова-Турчина (модель МТ)

Модель Н возникла как обобщение такого поведения, когда после предыдущего действия ясно, какое должно быть следующим (команда сама устанавливает следующую команду). Такое поведение типично для рутинных вычислений, на авто-

матизацию которых ориентировались первые компьютеры (они были предназначены для расчетов, связанных с созданием атомной бомбы).

Расчеты такого рода характеризуются данными относительно простой структуры – программы имеют дело с числами. Вся сложность поведения исполнителя определяется сложностью плана (то есть числом и связями указанных в нем действий). Управление последовательностью действий зависит от сравнения простых данных. Еще Джон фон-Нейман хорошо понимал, что для других классов применений могут потребоваться компьютеры, характеризующиеся другим типом поведения.

1.3.1. Перевод в польскую инверсную запись (ПОЛИЗ)

Рассмотрим, например, задачу перевода арифметической формулы в постфиксную форму. Другими словами, исходными данными для нашей программы должны быть обычные арифметические формулы, а в результате нужно получить их запись в ПОЛИЗе. Например:

(a+b)*(c+d) --> ab + cd + *.

Данные ко всякой программе записываются на некотором языке (являются знаками в некоторой знаковой системе). Чтобы их обработать, нужно воспользоваться правилами построения знаков в этой системе (синтаксисом языка) для распознавания структуры знака, затем семантикой знаковой системы, чтобы связать со структурой знака его смысл (денотат), и обработать данное в соответствии с его смыслом.

Пусть синтаксис языка формул, которые мы хотим обрабатывать, задают следующие правила БНФ:

```
<формула>::=<сумма> | <произведение> |<первичная>;
<сумма>::=<сумма>+<произведение> | <первичная>; <произведение>::=<произведение>*<-
первичная> |<первичная>;
<первичная>::=<число> | <переменная> | (<формула>).
```

Числа и переменные точно определять не будем, оставляя представление о них на интуитивном уровне (23 и 305 – числа; x, y, a, b, АЛЬФА – переменные).

Тогда 23 – формула (первичная, произведение, сумма), a+b*23 – также формула (сумма), (a+b)*23 – также формула (произведение); (a+*b) – не формула.

Семантика формул – общепринятая. Смыслом (денотатом) формулы будем считать число, получающееся из чисел, входящих в формулу, применением указанных операций в общепринятом порядке. Задача состоит в том, чтобы получить перевод в ПОЛИЗ, сохраняющий денотат (то есть в данном случае – над теми же числами нужно выполнить те же операции и в том же порядке).

Другими словами, было бы идеально, если бы вся программа записывалась фразой примерно такого вида:

```
перевод(<формула 1><операция><формула2>) =
   перевод(<формула 1 >)
```

```
перевод(<формула2>)
<операция>
```

Две ключевые абстракции – анализ и синтез. Можно заметить, что перевод текста с языка формул четко распадается на действия двух сортов – на распознавание компонент структуры исходной формулы и на компоновку ее образа в ПОЛИЗе из результатов перевода выделенных компонент. Когда действия этих сортов переплетены некоторым нерегулярным способом, то планировать, понимать, выполнять и проверять сложно. Чтобы уменьшить сложность, полезно выделить две ключевые абстракции (два понятия): анализ исходной структуры и синтез результирующей структуры, – и предложить знаковую систему для их взаимосвязанной конкретизации в рамках единой программы.

Модель Маркова. Тут уместно вспомнить язык нормальных алгоритмов Маркова (для единообразия назовем этот язык **моделью Маркова**).

Охарактеризуем эту модель с точки зрения нашей концептуальной схемы.

Базис: единственный скалярный тип данных – литера; единственная базисная операция – поиск-подстановка; единственная структура данных – строка (текст); единственная структура операций – цикл по подстановкам.

Развитие: явных средств нет. Только моделированием.

Дальнейший анализ модели можно предложить в качестве упражнения.

В модели Маркова анализ структуры встроен в исполнитель и управляется левой частью подстановки. Синтез структуры отделен от анализа – он управляется правой частью подстановки. Исполнитель распознает тривиальную структуру (слово), указанную слева, и заменяет ее столь же тривиальной структурой (словом), указанной справа.

С точки зрения нашей задачи модель Маркова недостаточно развита. Дело в том, что вид распознаваемых структур слишком тривиален. Хотелось бы приблизить средства описания вида структур, например, к БНФ. Шаги в нужном направлении сделаны в языке, созданном **В. Ф. Турчиным** в ИПМ АН СССР в 1966–1968 гг. и названном им «Рефал» (**ре**курсивных **ф**ункций **ал**горитмический язык). В основу Рефала положены три модификации модели Маркова.

1.3.2. Модификации модели Маркова (введение в Рефал)

Изложим модель языка Рефал, особенно интересного с точки зрения нашей концептуальной схемы потому, что он был задуман и реально используется как средство для эффективного определения других языков (другими словами, как базовый ЯП).

Первая модификация состоит в том, что в качестве (по-прежнему единственной) базисной структуры данных вместо произвольной строки (слова) используется «выражение» – строка, сбалансированная по скобкам.

Вторая модификация касается подстановки. Ее левая часть должна быть так называемым функциональным термом с возможными переменными. Правая

часть должна быть выражением, в котором можно использовать переменные из левой части подстановки (и только их).

Третья модификация касается поиска применимой подстановки. В отличие от модели Маркова, где заранее не фиксируется заменяемая часть обрабатываемого слова, в Рефале заменяемая часть обрабатываемого выражения фиксируется перед поиском применимой подстановки – это всегда так называемый ведущий функциональный терм.

Применимой считается подстановка с минимальным номером, левая часть которой согласуется с ведущим термом. Иными словами, применима подстановка с такой левой частью, где указан общий вид структуры (образец), частным случаем которого оказался ведущий терм.

Займемся теперь каждой из модификаций подробнее. Нам нужно уточнить смысл слов «выражение», «ведущий функциональный терм», «переменная» и «согласуется». Рассматриваемую модель ЯП назовем моделью **Маркова-Турчина (моделью МТ)**.

Строение выражений; поле зрения. Выделены три типа скобок – символьные (открывающая ‘ и закрывающая ’ кавычки), структурные (обычные круглые скобки) и функциональные (мы будем использовать фигурные скобки «{« и «}»).

Выражением называется всякая последовательность литер, сбалансированная по всем трем типам скобок; **термом** – выражение в скобках либо совсем без скобок; **символом** – отдельная литера либо последовательность литер в символьных скобках.

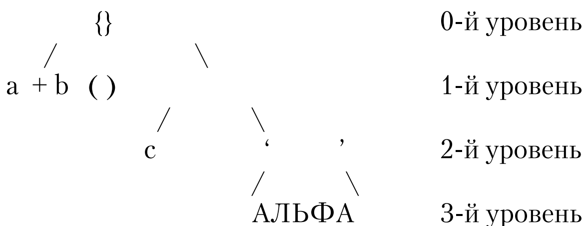
Например:

(a+b) – выражение	– структурный терм;
{a+b (c 'АЛЬФА')}	– выражение – функциональный терм;
'АЛЬФА'	– символ, терм, выражение;
}ab{	– не выражение.

По существу, выражение – это линейное представление дерева – структура этого вида часто используется в программировании именно потому, что наглядно воплощает идею иерархии, частичного порядка, пошаговой (последовательной) декомпозиции.

Дерево – это ориентированный граф (орграф) без циклов, в котором выделена вершина, называемая корнем дерева, и в каждую вершину, кроме корня, входит ровно одна дуга, причем из корня доступны все вершины. В дереве легко вводятся уровни иерархии (по длине пути из корня).

Так, выражение {a+b(c 'АЛЬФА')} может быть представлено деревом вида



Ведущим (функциональным) термом называется самый левый функциональный терм, не содержащий других функциональных термов. В примерах ведущие термы выделены:

(a+b{c+d});
{ АЛЬФА (a*b)}{cd}xl0
(100 DO 3 {I={1}(,3)}).

Таким образом, мы полностью описали допустимую структуру поля зрения МТ-исполнителя (МТ-машины). В этом поле помещается обрабатываемый объект, который может быть только выражением. В качестве очередной заменяемой части всегда выбирается ведущий терм. Если такового нет, то делать исполнителю нечего, и он останавливается.

Выражение, оставшееся в поле зрения, считается результатом выполнения программы, находящейся в поле определений исполнителя.

В авторской терминологии это поле называется «поле памяти». Так говорить нам неудобно. В модели Н и программа, и данные находились в памяти. Естественно считать, что поле зрения МТ-машины – также часть памяти. Термин «поле определений» лучше отражает суть дела.

Поле определений; МТ-предложения. Мы изучаем модели ЯП. Поэтому будем позволять себе «вариации на тему» рассматриваемого языка-прототипа, когда такие вариации упрощают рассмотрение. Например, говоря о МТ-предложениях, мы не будем строго следовать их авторской трактовке.

В модели Маркова средства описания правил анализа и синтеза бедны – можно лишь явно выписывать заменяемое и заменяющее подслова (левую и правую части марковской формулы соответственно).

Основная идея обобщения марковской формулы состоит в том, чтобы за счет введения локальных переменных наглядно изображать одной (обобщенной) формулой сразу целый класс подстановок (применимых к функциональным термам определенной структуры).

Ключевыми понятиями при этом служат интерпретация переменных и согласование (терма с обобщенной подстановкой при определенной интерпретации ее переменных).

Интерпретация переменных – это функция типа I:N->N, где N – множество обозначений переменных. V – множество их допустимых значений.

Интерпретация напоминает состояние в модели Н. Только вместо адресов – обозначения переменных. Это, по сути, одно и то же. Но называем мы их по-разному, так как они играют разные роли. Состояние в модели Н – глобальный объект, сохраняющийся между последовательными операциями, а интерпретация в модели МТ – локальный объект, действующий внутри операции подстановки.

При конкретной интерпретации переменных обобщенная подстановка (в Рефале ее называют **предложением**, или **Рефал-предложением**) изображает конкретную марковскую формулу подстановки.

Например, предложение

{10 e 00 s 1} -> s 101 e,

где e и s– (локальные) переменные, при интерпретации

i1={e->00, s->11)

(здесь фигурные скобки – обозначение множества пар, составляющих интерпретацию) изображает марковскую формулу

{100000111}-->1110100,

а при интерпретации

i2={e->ABC,s->D} –

марковскую формулу

{10ABC00D1}-->D101ABC.

Соответственно, левая часть предложения изображает левую часть марковской формулы, а правая часть предложения – правую часть формулы.

Согласование – это тройка (t,i,s), где t – ведущий терм, s – предложение и i – такая интерпретация, при которой левая часть s изображает t.

Итак, за счет различных интерпретаций переменных одна обобщенная марковская подстановка (предложение) способна изображать целый класс марковских подстановок (что и требовалось).

Однако этот класс не должен быть слишком широким. Ведь каждое предложение должно быть приспособлено для наглядного изображения вполне определенного содержательного преобразования поля зрения. Поэтому следует принять меры к тому, чтобы, во-первых, изображаемые подстановки не нарушали структуру поля зрения и, во-вторых, чтобы можно было управлять допустимыми значениями переменных (другими словами, управлять их типом).

Наконец, в-третьих, необходимо установить такие правила согласования предложения с ведущим термом, чтобы анализ и синтез были однозначными. Так что правила согласования должны обеспечивать единственность подразумеваемой программистом согласующей интерпретации (при фиксированном поле зрения).

Первое и второе достигаются за счет ограничений на класс допустимых интерпретаций, третье – за счет ограничений на класс допустимых согласований.

Допустимые интерпретации должны удовлетворять двум условиям:

- значения переменных, а также обе части изображаемой подстановки должны быть выражениями (так что МТ-преобразования не выводят за класс выражений);
- значение переменной должно соответствовать **спецификатору**, который указывается непосредственно после обозначения переменной и отделяется двоеточием «:».

Понятие спецификатора связано с еще одним (в некотором смысле ортогональным) направлением обобщения марковской формулы подстановки. Это направление мы оставим открытым и будем использовать пока только очень простые спецификаторы. Именно в качестве спецификатора можно написать «сим-

вол» или «терм» (это значит, что значениями переменной могут быть только символы (только термы)) или в круглых скобках можно явно перечислить допустимые значения переменной. Например, s: символ – переменная, значениями которой могут быть только символы, t:терм – только термы, s:(+**I**-) – значениями s могут быть только литеры «+» или «–».

Ограничения на согласования состоят в том, что допустимыми считаются только так называемые ориентированные согласования. Они бывают **левыми** или **правыми**.

Определим **левое (лево-ориентированное) согласование**. Правое определяется по симметричным правилам.

Будем называть переменную y1 в функциональном терме **левой** для переменной y2, если самое левое вхождение y1 расположено левее самого левого вхождения переменной y2.

Будем говорить, что согласование (t,i’,s) **короче** согласования (t,i,s), если в t найдется переменная y1, для которой i’(y1) короче i(y1), причем для любой переменной z, левой для y1 в терме t, i’(z) совпадает с i(z).

Согласование (t,i,s) называется **левым**, если оно самое короткое из возможных согласований t и s.

Таким образом, основная идея левого согласования – левые переменные при поиске согласующей интерпретации удлиняются в последнюю очередь.

По умолчанию предполагается, что допустимы только левые согласования. Допустимость только правых согласований указывается буквой R после закрывающей функциональной скобки в левой части предложения.

Например, предложение

{e1+e2}-->{e1}{e2}+

согласуется с термом {a+b+c+d} интерпретацией

{e1-->a, e2-->b+c+d}

и изображает формулу подстановки

{a+b+c+d} --> {a}{b+c+d}+,

а предложение

{e1+e2}R --> {e1}{e2}+

согласуется с тем же термом интерпретацией

{e1-->a+b+c, e2-->d}

и изображает формулу подстановки

{a+b+c+d} --> {a+b+c}{d}+.

В Рефале принимаются меры к тому, чтобы всегда можно было отличить переменные от постоянных частей предложения. Если есть опасность спутать переменную и постоянную, то постоянную будем выделять.

Подводя итог, можно сказать, что **идея подстановки** работает в Рефале три раза:

1) интерпретация i определяет подстановку значений переменных вместо их обозначений;

2) тем самым она определяет соответствие обобщенной и конкретной марковских подстановок (то есть «подстановку» конкретной подстановки вместо обобщенной);
3) правая часть этой конкретной подстановки заменяет ведущий терм.

При этом подбор согласующей интерпретации есть, по существу, анализ ведущего терма, а порождение конкретной правой части подстановки при найденной интерпретации – синтез заменяющего выражения (в правой части всегда должно быть правильное выражение – это еще одно требование Рефала). В этом смысле левая часть предложения служит **образцом структуры ведущего терма** (терм и предложение согласуются, если структура терма соответствует образцу), а правая – **образцом для синтезируемого заменяющего выражения**.

Упражнение. Покажите, что если ведущий терм согласуется с некоторым предложением, то соответствующее согласование единственно.

Подсказка. Оно либо левое, либо правое.

1.3.3. Исполнитель (МТ-машина)

Теперь легко объяснить, как действует исполнитель, имея в поле зрения обрабатываемое выражение, а в поле определений – программу (то есть кортеж предложений). Он выполняет следующий цикл:

1) выделяет ведущий терм. Если такового нет, останавливается. Выражение в поле зрения считается результатом;
2) ищет первое по порядку предложение, которое согласуется с ведущим термом. Соответствующее согласование всегда единственно. Значит, единственна и изображаемая при соответствующей интерпретации переменных марковская подстановка. Она и применяется к ведущему терму. И цикл начинается сначала с обновленным полем зрения.

Если нет согласующихся с ведущим термом предложений, то исполнитель останавливается с диагностикой «согласование невозможно».

1.3.4. Программирование в модели МТ

Задачу перевода в ПОЛИЗ (с учетом старшинства операций) решает следующая программа:

```
{e1+e2}R --> {e1}{e2}+
{e1*e2}R --> {e1}{e2}*
{(e)} -> {e}
{e} -> e
```

Упражнение 1. Доказать, что это правильная программа.
Обратите внимание, действиями исполнителя полностью управляет структура обрабатываемых данных.

Упражнение 2. Можно ли эту программу написать короче? Например, так:

```
{e1 s:(+I*) e2}R -> {e1}{e2}S
{(e)} -> {e}
{e} -> e
```

Упражнение 3. Можно ли здесь отказаться от правого согласования?

Упражнение 4. Напишите программу аналитического дифференцирования многочленов по переменной «x».

1.3.5. Основное семантическое соотношение в модели МТ

Рассмотрим функцию sem, реализуемую МТ-программой p. Ее тип, очевидно:

sem:P x E -> E,

где P – программы, E – выражения.

Уже тип функции sem указывает на принципиальное отличие от модели Н – программа не меняется. В модели Н программа – часть (изменяемого) состояния.

Пусть ft – функция, выделяющая в выражении ведущий функциональный терм, l и r – функции, выделяющие соответственно левую и правую части выражения, оставшиеся после удаления ведущего терма. Конкатенацию (соединение) строк литер будем обозначать точкой «.». Удобно считать, что если ведущего терма в выражении e нет, то ft = <>, r(e) = e, где <> обозначает пустое слово. Все эти три функции типа E -> W, где W – тип «слов» (произвольных последовательностей литер), так как результаты могут и не быть выражениями.

Пусть далее step – функция типа

P x T' -> E,

где T' = T U {<>}. Она реализуется одним шагом работы МТ-машины – отображает пару (программа, ведущий терм или пусто) в выражение, получающееся из этого терма применением соответствующей МТ-подстановки. Функция step, естественно, частичная – она не определена, если согласование с p невозможно; step(p,<>) = <> по определению.

Учтем, что p не меняется и вся зависимость sem от p скрыта в функции step. Поэтому позволим себе для краткости явно не указывать p среди аргументов функций sem и step. Тогда можно выписать следующее соотношение для sem:

sem(e) = sem(l(e).step(ft(e)).r(e)).

Если обозначить l(e), r(e) и ft(e) соответственно через l, r и f, то получим более выразительное соотношение:

(a) sem(l.ft.r) = sem(l.step(ft).r).

Покажем, что на самом деле справедливо следующее основное соотношение

(b) sem(l.ft.r) = sem(l.sem(ft).r).

Действительно, если step(ft) не содержит функциональных термов, то

sem (ft) = step(ft)

и (b) следует из (a).

Если же step (ft) содержит функциональные термы, то так как l таких термов не содержит, все функциональные термы из step (ft) будут заменены раньше, чем изменится l или r. Но последовательные замены термов в step (ft) – это и есть вычисление sem(ft).

Если такое вычисление завершается и между l и r не остается функциональных термов, то вычисление sem от исходного выражения будет нормально продолжено с выражения l.sem (ft).r.

Если же sem(ft) вычислить не удается из-за отсутствия согласования, то на этом же месте окажется невозможным согласование и для исходного выражения. Тем самым равенство доказано.

В соотношении (b) зафиксированы следующие свойства МТ-семантики:

1. Результат применения программы к ведущему терму не зависит от его контекста, а значит, и от истории применения программы к исходному выражению.
2. «Область изменения» в выражении e до полного вычисления его ведущего терма ограничена этим термом.
3. Если l и r не содержат функциональных скобок, они никогда не могут быть изменены.

Аналогичными рассуждениями можно обобщить соотношение (a). Обозначим через ft1,...,ftn последовательные терминальные функциональные термы в e (то есть не содержащие других функциональных термов), а через r0,...,rn – слова, не содержащие функциональных термов и такие, что

e = r0.ft1.,...,.ftn.rn.

Тогда справедливо следующее соотношение:

(c) sem(r0.ft1.,...,.ftn) –
sem (r0.sem (ft 1).,...,.sem (ftn).rn).

Упражнение. Докажите справедливость этого соотношения.
Не забудьте, что участок

r0.sem(ft1).,...,.sem(ftn).rn

может содержать функциональные термы.

Отметим также очевидное соотношение

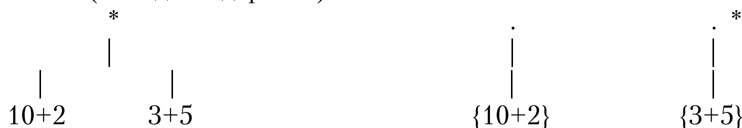
sem(sem(e)) = sem(e).

Таким образом, обработка в модели МТ обладает четкой иерархической структурой. Другими словами, выполнение программы p над выражением e можно представлять себе как «вычисление» этого выражения, начиная с любого из «терминальных функциональных поддеревьев» соответствующего дерева.

1.3.6. Пример вычисления в модели МТ

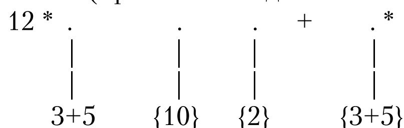
Сопоставим вычисление по школьным правилам выражения (10+2)* (3+5) с обработкой в модели МТ выражения {10+2} {3+5}* по программе перевода в ПОЛИЗ. Изобразим последовательно получаемые деревья, соответствующие обрабатываемым выражениям (слева – для школьной арифметики, справа – для модели МТ).

Шаг 1 (исходные деревья)



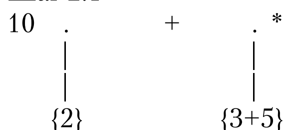
Деревья явно похожи (вершины изображают операции, дуги – отсылки к тем операндам, которые еще следует вычислить).

Шаг 2 (применение одной из операций, для которых готовы операнды)

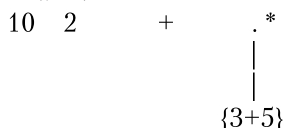


Видно, что дерево справа «отстает» от дерева слева. Сказывается различие результатов функций step и sem. Последим за правым деревом до завершения вычисления функции sem ({10+2}).

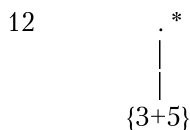
Шаг 2.1



Шаг 2.2



Шаг 2.3



Вот теперь деревья снова похожи!

Слово «вычисление» означает здесь процесс, вполне аналогичный вычислению значения обычного школьного алгебраического выражения после подстановки вместо переменных их значений. Однако аналогия касается не типа допустимых значений (в школьной алгебре – числа, а здесь – сбалансированные по скобкам тексты), а способа планирования обработки (способа программирования).

И в школьной алгебре, и в модели МТ план обработки в определенном смысле содержится в обрабатываемом (вычисляемом) выражении. Роль исполнителя состоит в том, чтобы выполнять указанные операции над допустимыми операндами, подставляя результат операций в обрабатываемое выражение на место вычисленного терма.

Существенные отличия состоят в том, что, во-первых, школьные операции считаются заранее известными, предопределенными, а смысл единственной МТ-операции step задается полем определений; во-вторых, результат школьных операций – всегда окончательный (новых операций в нем не содержится – это число), а результат операции step – в общем случае «промежуточный»; им может оказаться выражение с новыми функциональными термами. Заметим, что второе отличие исчезает, если от функции step перейти к функции sem – ее результат всегда «окончательный», ведь (sem(sem(e))=sem(e)).

```
Шаг 3
12 * 8       10  2   +        .       .       + *
                              |       |
                              |       |
                             {3}     {5}

Шаг 3.1
12 * 8       10  2  +         3       . + *
                                      |
                                      |
                                     {5}

Шаг 3.2
12 * 8       10  2  +  3  5  + *

Шаг 4
96           10  2 +  3  5  + *
             (нет функциональных термов)
```

Итак, мы убедились, что МТ-вычисления очень похожи на вычисления обычных арифметических формул.

Несколько замечаний. Вычисления по формулам очень поучительны для программистов. Из этого древнейшего способа планирования вычислений можно извлечь много полезных идей.

Во-первых, это четкая структура вычислений – она, как мы видели, древовидная.

Во-вторых, операнды рядом с операциями (их не нужно доставать из общей памяти).

В-третьих, результат не зависит от допустимого изменения порядка действий (с сохранением иерархии в соответствии с деревом выражения). Отсюда – путь к параллельному вычислению, если позволяют вычислительные ресурсы (когда есть несколько процессоров).

В-четвертых, принцип синхронизации таких вычислений прост – всякая операция должна ждать завершения вычислений своих операндов (ничто другое на ее выполнение не влияет). На этом принципе основаны так называемые конвейерные вычисления и вычисления, «управляемые потоком данных» (data flow).

В-пятых, результаты операций никуда не нужно посылать – они нужны там, где получены.

Наконец, отметим еще одну идею, в последние годы привлекающую внимание исследователей, стремящихся сделать программирование надежным, доказательным, систематическим. Речь идет о том, что над школьными формулами можно выполнять систематические преобразования (упрощать, приводить подобные члены, явно выражать неизвестные в соотношениях и т. п.). Есть надежда определить практичную алгебру преобразований и над хорошо организованными программами. Это позволит систематически выводить программы, проверять их свойства, оптимизировать и т. п.

Обратите внимание, значение функции sem не зависит от порядка вычисления терминальных функциональных термов. А в нашем исходном определении модели МТ требовалось, чтобы всегда выбирался самый левый из всех таких термов. При отсутствии взаимного влияния непересекающихся термов это требование несущественно. В реальном Рефале указанное влияние возможно.

1.3.7. Аппликативное программирование

Модель МТ относится к широкому классу **аппликативных моделей вычислений**. Это название (от слова *apply* – применять) связано с тем, что в некотором смысле единственной операцией в таких моделях оказывается операция **применения** функции к ее аргументу, причем единственной формой влияния одного применения на другое служит связь по результатам (суперпозиция функций). В частности, функции не имеют побочного эффекта.

> Напомним, что побочным эффектом функции называется ее влияние на глобальные объекты, не являющиеся аргументами; в модели МТ переменные локальны в предложениях, а отсутствие побочного эффекта на поле зрения мы уже обсуждали.

Аппликативные модели привлекательны тем, что сохраняют многие полезные свойства вычислений по формулам. Самое важное из них – простая и ясная структура программы, четко отражающая требования к порядку вычислений и связям компонент. Вместе с тем по своей алгоритмической мощности аппликативные модели не уступают другим моделям вычислений.

Задача. Доказать, что модель МТ алгоритмически полна, то есть для всякого нормального алгоритма А найдется эквивалентная ему МТ-программа (допускается заменять алфавит, в котором работает А).

Пока наша модель МТ бедна в том отношении, что ко всем термам применяется одна и та же функция step. Это плохо и потому, что программу трудно понимать (особенно если она длинная), и потому, что она будет медленно работать, если каждый раз просматривать все предложения поля определений. К счастью, модель МТ легко приспособить к более гибкому стилю аппликативного программирования.

1.3.8. Структуризация поля определений. МТ-функции

Допустим, что имеется неограниченный набор различных функциональных скобок (**как это можно обеспечить?**). Будем группировать предложения, записывая подряд друг за другом такие предложения, левая часть которых заключена в одинаковые функциональные скобки.

Тогда ведущий терм будет однозначно указывать на соответствующую группу предложений (в ней и только в ней достаточно искать согласование).

В этом случае функция step распадается на отдельные функции, а программа – на определения этих функций (за что соответствующее поле, где помещается МТ-программа, мы и назвали полем определений).

Достаточно различать только левые функциональные скобки (**почему?**).

Будем считать левой функциональной скобкой название (идентификатор) функции вместе с непосредственно следующей за ним открывающей фигурной скобкой.

Например, программу перевода в ПОЛИЗ запишем так:

```
перевод{e1+e2}R -> перевод{e1}  перевод{e2} +
перевод{e1*e2}R -> перевод{e1}  перевод{e2} *
перевод{(e)} -> перевод{e}
переводе{e} -> e.
```

Эту совокупность подстановок естественно считать определением МТ-функции «перевод». Его удобно использовать в большой программе среди других подобных определений.

Поле зрения с исходными данными для перевода может иметь при этом вид

перевод {(a+b) * (c+d)}.

Так что и запись самой программы в модели МТ, и обращение к ней весьма напоминают то, что мы выбрали в качестве идеала в самом начале разговора об анализе и синтезе.

Недостаточна, правда, выразительная сила применяемых в нашей модели образцов. Поэтому приходится писать подробнее, чем в БНФ.

До сих пор поле определений рассматривалось как определение одной функции. Это была либо функция step, если результат считался полученным после одного применения подстановки, либо (в общем случае рекурсивная) функция sem, если результатом признавалось только выражение без функциональных термов.

Когда поле определений разбито на группы подстановок с одинаковыми левыми функциональными скобками, каждую такую группу естественно считать определением отдельной функции. С точки зрения одного шага МТ-машины, это функция, представляющая собой сужение функции step на ведущие термы с конкретной функциональной скобкой. С технологической точки зрения (с точки зрения программиста), это рекурсивная МТ-функция, представляющая собой сужение функции sem на те же термы.

> **Замечание.** Применение МТ-функций предполагает уже некоторый элемент прогнозирования со стороны программиста и контроля со стороны МТ-машины, отсутствовавший в исходной модели.
>
> Употребляя конкретную функциональную скобку в правой части предложения, программист прогнозирует, что при определенном поведении исполнителя (если будет выбрано именно это предложение) потребуется определение соответствующей функции.
>
> МТ-машина, со своей стороны, получает возможность просмотреть поле определений и проверить, что в нем присутствуют определения всех использованных МТ-функций. Другими словами, становится возможным статический контроль программ (то есть контроль программ до их выполнения, без учета исходных данных).

Итак, мы можем определять в программе столько рекурсивных функций, сколько нужно.

Вот, например, как выглядит программа аналитического дифференцирования, в которой используется частная производная по x и частная производная по y:

```
Dx{e1+e2}R -> Dx{e1} + Dx{e2}
Dx{e1*e2}R -> e1*(Dx{e2}) + e2*{Dx{e1})
Dx{(e)} -> Dx{e}
Dx{'x'} -> 1
Dx{s: символ} -> 0
Dy{el+e2}R -> Dy{el} + Dy{e2}
......
......
Dy{'y'} ->1
Dy{s: символ} -> 0
```

> **Задача.** Можно ли объединить эти функции? Как это сделать?

Глава 2

Функциональное программирование (модель Б)

2.1. Функциональное программирование в модели МТ 290
2.2. Функциональное программирование в стиле Бэкуса (модель Б) 299

2.1. Функциональное программирование в модели МТ

В соответствии с определением А. П. Ершова **функциональное программирование** – это способ составления программ, в которых единственным действием является вызов (применение) функции, единственным способом расчленения программ на части – введение имени для функции и задание для него выражения, вычисляющего значение этой функции, единственным правилом композиции (структурой операций) служит суперпозиция функций.

Ясно, что модель МТ с учетом последней «функциональной» модификации позволяет программировать в строго функциональном стиле. Другими словами, это одна из моделей функционального программирования.

> Таким образом, одно из отличий «функционального» программирования от «аппликативного» – возможность явно определять (в общем случае рекурсивные) функции.

Дополнительные примеры программирования в «функциональном стиле» мы приведем чуть позже, а пока дадим краткий обзор «функциональной» модели МТ с точки зрения нашей концептуальной схемы.

2.1.1. Модель МТ с точки зрения концептуальной схемы

Базис: скалярные данные – только литеры, скалярные операции – только обобщенная поиск-подстановка. Структурные данные – только выражения (есть подтипы: символ и терм), структурные операции – встроенный цикл, легко приводящий к комбинациям функций.

> Говорят, что функции комбинируются горизонтально, если их результаты являются непосредственными составляющими одного функционального терма.
> Говорят, что функции комбинируются вертикально, если одна из них не может быть вычислена до завершения вычисления другой. В такой комбинации первая называется внешней, а вторая – внутренней.
> В модели МТ применяются и горизонтальная, и вертикальная комбинации функций. Горизонтальная комбинация называется также конструкцией, а вертикальная, при которой результат внутренней служит полным аргументом внешней, – композицией; произвольная комбинация – суперпозицией.

Развитие: вверх – только функции типа E -> E (однако за счет структурированности выражений это весьма мощное средство развития (как будет показано)); вниз – средств нет.

Защита: в базисе средств нет.

2.1.2. Модель МТ и Лисп

Можно показать, что модель МТ отражает не только свойства такого реального языка, как Рефал, но и свойства еще одного заслуженного языка – языка Лисп, созданного Джоном Маккарти в 1960 г. и с тех пор прочно удерживающего позиции одного из самых распространенных ЯП (особенно в качестве инструментального языка в области искусственного интеллекта). В последние годы интерес к нему усилился еще и как к первому реальному языку функционального программирования.

Единственной базисной структурой данных в Лиспе служит список – так называемое **S-выражение**. Оно естественно представимо в модели МТ выражением в круглых скобках. Элементарные **селекторы** и **конструкторы** Лиспа (предопределенные функции, позволяющие выбирать из списков компоненты и строить новые списки из заготовок) легко программируются в модели МТ.

> Приведем упрощенные определения МТ-функций, способных играть роль селекторов и конструкторов. Для краткости всюду ниже будем считать, что с обозначениями МТ-переменных, начинающихся с букв s и t, связаны соответственно спецификаторы «символ» и «терм» (так и делается в реальном Рефале).

Выбор головы (первого элемента) списка:

```
первый {(t e)} -> t.
```

Выбор хвоста списка:

```
хвост {(t e)} -> (e).
```

Конструирование (создание) списка:

```
создать {e}  -> (e).
```

Соединение списков:

```
соединить {(e1)(e2)} -> (e1 e2).
```

Подобным образом программируются и другие функции, аналогичные примитивам Лиспа.

> **Упражнение.** Аккуратно выпишите МТ-определения примитивов (базисных функций) Лиспа. Учтите все их тонкости. Рассмотрите отличия функций «первый», «хвост» и «создать» от функций car, cdr и cons Лиспа.

Обратите внимание, по существу, мы продемонстрировали способность модели МТ к развитию – довольно легко определить в модели МТ новый язык, аналогичный Лиспу.

2.1.3. Критерий концептуальной ясности и функции высших порядков

Продолжая рассматривать модели ЯП с технологической позиции, продемонстрируем технологическую потребность в функциях высших порядков (то есть функциях, аргументами и (или) результатами которых служат функции). Затем

покажем, как их можно ввести в модели МТ, и рассмотрим модель Бэкуса (**модель Б**), в которой функции высших порядков играют ключевую роль (введены в базис).

Напомним, что к модели МТ мы пришли от идеи разделения анализа и синтеза в обработке данных. И получили мощные средства развития, как только ввели удобную базисную структуру данных (выражение), локализовали область воздействия на эту структуру (ведущий терм) и упростили отбор возможных воздействий (ввели МТ-функции).

Теперь у нас в руках аппарат, который можно развивать в различных направлениях и (или) использовать в различных целях.

Например, в реальном Рефале введены операции, позволяющие изменить поле определений в процессе исполнения программы. Это так называемые операции «закапывания» и «выкапывания» определений по принципу магазина. При таком развитии получается стиль программирования, более близкий к традиционному, с присваиванием глобальным переменным и взаимным влиянием непересекающихся термов. Нас здесь больше интересует развитие в функциональном стиле.

Воспользуемся аппаратом развития, чтобы показать богатейшие возможности функционального программирования с точки зрения достижения концептуальной ясности программ.

Идеалом будет служить такая программа, в которой в некотором смысле **нет ничего лишнего**.

Другими словами, этот **критерий концептуальной ясности** можно выразить так: **структура функции, реализуемой программой, совпадает со структурой программы.**

Однако при этом функция «состоит» из соответствий, а программа – из операций.

Важнейшая абстракция, способствующая приближению к намеченному идеалу, – функция высшего порядка (или, как мы ее назовем, следуя Бэкусу, **форма**). Ближайшая задача – показать это на достаточно убедительных примерах.

Замечание. Важно понимать, что хотя модель МТ, конечно, алгоритмически полна, она (как и любая другая модель) не универсальна в том смысле, что в ней не всегда легко вводить любые абстракции. Однако формы в ней вводить довольно легко.

2.1.4. Зачем нужны функции высших порядков

Функции высших порядков возникают совершенно естественно. Классический пример – программа интегрирования (вычисления определенного интеграла). Она реализует некоторую форму, аргументом которой служит подынтегральная функция, а результатом – число. Программа аналитического дифференцирования реализует форму, аргументом которой служит некоторая функция (заданная, например, многочленом), а результатом – ее производная, то есть снова функция.

Любая из рассмотренных нами функций, выражающих денотационную семантику модели Н или МТ, получается, как мы видели, определенной комбинацией исходных функций, соответствующих базисным конструкциям. Если изменить эти исходные функции, не меняя зафиксированной нами формы, представленной их комбинацией, то получим другую семантику модели.

Так, при изменении в модели Н семантики операций изменится семантика программы. В модели МТ также можно варьировать, например, правила согласования или подстановки без всякого изменения денотационных соотношений – они-то и фиксируют вполне определенную форму, отображающую пару (step,p) в sem.

Замечания о функциях высших порядков. Напомним, что мы рассматриваем только функции типа

E -> E.

В частности, это означает, что все они формально имеют один аргумент. Фактически может быть столько аргументов, сколько нужно, – ведь аргументами можно всегда считать последовательные термы выражения. Отдельные аргументы можно всегда заключить в круглые скобки.

Однако, чтобы не загромождать примеры, договоримся, что отделение аргументов пробелами эквивалентно заключению в скобки. Другими словами, будем в значительной степени абстрагироваться от «проблемы круглых скобок», концентрируя внимание на принципиальных моментах (хорошо понимая, что в практическом программировании от этой проблемы никуда не деться – в Лиспе, например, она одна из самых неприятных).

Как только мы сказали, что имеем дело только с функциями типа

E -> E,

сразу возникает вопрос: как же быть с формами. У них-то аргументы – функции, а не выражения. Ответ состоит в том, что и аргументы, и результаты форм всегда будут представлены некоторыми выражениями (например, символами – названиями функций).

Примем стиль изложения, при котором смысл вводимых программистских абстракций будем объяснять с помощью определений в модели МТ. Иногда это может показаться трудным для восприятия. Зато мы, во-первых, постоянно упражняемся в программировании в модели МТ; во-вторых, немедленно демонстрируем конкретизацию вводимой абстракции, а именно ее реализацию в известной модели.

> Такой стиль изложения можно назвать проекционным – вместе с новым понятием излагается его проекция (перевод) на уже известный инструментальный язык. В нашем случае основу этого языка предоставит модель МТ.

Первая форма, которую следовало бы рассмотреть, – это, конечно, **аппликация** (обозначим ее двоеточием «:»). Она применяет указанную в ее аргументе функцию (возможно, форму) к остальным компонентам аргумента. Можно было бы определить аппликацию в общем виде, однако нам удобнее считать, что определе-

ние МТ-функции «:» формируется постепенно. А именно группа предложений со специальной функциональной скобкой вида «:{« пополняется новыми предложениями по мере введения новых форм.

Таким способом (за счет возможностей МТ-образцов) можно определять новые формы (и обычные функции), не требуя, чтобы обращение к ним было обязательно префиксным (то есть чтобы название функции предшествовало аргументам). Префиксный способ требует слишком много скобок, поэтому его желательно избегать, когда функция (форма) обладает, например, свойством ассоциативности.

Упражнение. Покажите, как можно вводить инфиксные функции.

Подсказка. Вспомните о переводе в ПОЛИЗ.

Пока будем считать, что в группе аппликации (апл) лишь два предложения

```
(апл)   :{(f) e} -> :{f e}
        :{s_f e} -> s_f{ e },
```

где f – переменная, обозначающая вызов некоторой формы, а s_f – переменная, обозначающая название применяемой МТ-функции.

Первое предложение снимает скобки, ограничивающие вызов формы (они могли остаться после вычисления значения ее результата, если он был задан инфиксным выражением), а второе выписывает функциональный терм, который служит вызовом применяемой функции.

Подразумевается, что определения применяемых функций в МТ-программе имеются. Предложения (апл) будут оставаться последними в группе аппликации. Новые будем добавлять в ее начало (чтобы сначала действовали формы, а лишь затем их результаты – обычные функции).

2.1.5. Примеры структурирующих форм

Намеченный идеал концептуальной ясности наводит на мысль, что наиболее важными могут оказаться формы, помогающие рационально структурировать программу – выражать ее смысл (реализуемую функцию) простой и понятной комбинацией других функций. Рассмотрим несколько таких структурирующих форм.

1. **Композиция** (ее часто обозначают звездочкой «*»). Применить результат композиции двух функций f и g – значит применить функцию f к результату применения g. «Применить» – это значит использовать аппликацию. В модели МТ определение композиции выглядит так:

   :{(f*g)e} -> :{(f) :{(g) e}}.

   Точнее говоря, чтобы это предложение заработало как определение новой формы (а именно композиции), им следует пополнить группу (апл) из предыдущего пункта.
2. **Общая аппликация** (применение указанной в аргументе функции ко всем непосредственным составляющим обрабатываемого выражения). Обозначим ее через «A» по аналогии с квантором всеобщности. Для ее определения через аппликацию в группу (апл) следует добавить два МТ-предложения

:{(Af)t e} -> :{(f)t} : {Af)e}
:{(Af) } -> <> .

Итак, указанная выражением f функция применяется к компонентам обрабатываемого выражения. Получается выражение, составленное из результатов всех применений.

Вопрос. Зачем понадобилось второе предложение?

3. **Конструкция** (ее обозначим запятой «,»). Применить результат конструкции двух функций f и g к выражению e – значит получить конкатенацию выражений f(e) и g{e}.
   Определить конструкцию в модели МТ можно так:
   :{(f,g) e} -> :{(f)e} :{(g)e}.
4. **Редукция**, которую обозначим через «/». Название, идея и обозначение восходят к Айверсону, автору языка Апл – одного из самых распространенных диалоговых языков. Своей исключительной лаконичностью этот язык в значительной степени обязан функциям высших порядков:
   :{(/f) t1 t2 e } -> :{(f) t1 :{(/f) t2 e}}.
   :{(/f) t } -> t.
   Идея редукции – в том, что бинарная операция f (двухместная функция) последовательно применяется, начиная с конца выражения вида (t1 t2 e), то есть выражения, в котором не меньше двух составляющих. Название этой формы подчеркивает, что обрабатываемое выражение сворачивается к одному терму (редуцируется) за счет последовательного «съедания» пар компонент выражения, начиная с его конца.
   Например, с помощью редукции можно определить функцию «сумма»:
   сумма{e} -> :{(/+) e}
   Тогда если считать, что бинарная операция «+» предопределена и ее можно использовать префиксным способом, получим
   сумма{10 20 30} = :{(/+) 10 20 30} =
   =:{+10 :{(/+) 20 30}} =
   = :{+10 :{+20 :{(/+) 30}}} =
   = :{+10 :{+20 30}} = :{+10 50} = 60 .
   Обратите внимание, насколько прост и привычен вид программы-формулы
   сумма{10 20 30} = 60.

Итак, мы определили конструкцию, общую аппликацию, композицию, редукцию. В том же стиле с помощью аппликации можно определить и другие полезные формы. Если программировать с использованием таких форм (и некоторых других), то по существу мы будем работать в модели Бэкуса (**модели Б**). И снова развитие МТ-модели новыми функциями дает новый язык – язык Бэкуса.

Отличительная черта модели Бэкуса – **фундаментализация идеи функциональных форм**. В частности, четыре названные выше формы считаются примитивными (предопределенными, то есть определенными средствами, выходящими

за рамки модели). Аналогичная идея – одна из основных в языке Апл Айверсона. Однако Айверсону, в отличие от Бэкуса, не удалось ее фундаментализировать (выделить как важнейшую, как основу целого направления в программировании).

Определим в модели МТ еще несколько функций, полезных для работы с выражениями.

```
реверс{t e} -> реверс{e} t .
реверс{ } -> <>.
```

Эта функция преобразует выражение вида

t1 ... tn

в выражение вида

tn ... t1,

где ti – термы.

Следующая функция – транспонирование (для краткости будем обозначать ее «транс»). По сути дела, как показывают следующие примеры, это обычное транспонирование матриц.

```
транс{(a b c) (k 1 m)} = (a к) (b l) (c m)
транс{(a b) (c k (l m)} = (a c l) (b k m)
транс{(a b c) (k l m) (o p r)} = (a к o) (b l p) (c m r)
```

Здесь строки матрицы представлены последовательными термами МТ-выражения (это обычный способ представления структур в Рефале).

Определим теперь функцию «транс» точно:

```
транс{e} -> первые{e} транс{хвосты{e}}.
транс{} -> <>,
```

где «первые» – функция, выделяющая список первых элементов последовательных подвыражений, а «хвосты» – функция, выдающая список хвостов от последовательных подвыражений. Представим сначала их действие примерами.

```
первые{(a b c) (k 1 m)} = (a k)
первые{ (a b) (c k) (1 m)}= (a c l)
первые{ (a b c) (k l m) (o p r)} = (a k o)
хвосты{(a b c) (k l m)} = (b c) (l m)
хвосты{(a b) (c k) (l m))= (b) (к) (m)
хвосты{((a b) c) ((r l) m)} = (c) (m)
```

Теперь определим эти функции точно:

```
первые{(t1 e1) e2} -> (t1 первые{e2} )
первые{ } -> <>.
хвосты{(t1 t2 e1) e2} -> (t2 e1) дл-хвосты{e2} .
хвосты{(t1) e} -> кор-хвосты{e} .
дл-хвосты{(t1) t2 e1) e2} ->
                        (t2 e1) дл-хвосты{e2} .
дл-хвосты{ } -> <>.
кор-хвосты {(t) e} -> кор-хвосты{e} .
кор-хвосты{ } -> <>.
```

Вопрос. Для чего понадобилось вводить функции дл-хвосты и кор-хвосты?

Подсказка. Мы хотим транспонировать только матрицы.

2.1.6. Пример программы в стиле Бэкуса

Теперь можно написать программу, выражающую в некотором смысле «идеал» программирования в стиле Бэкуса. Точнее, мы напишем программу-формулу, вычисляющую скалярное произведение двух векторов. Будем действовать методом пошаговой детализации.

Допустим, что предопределены функции «сложить» (+) и «умножить» (×). Представим подлежащие перемножению векторы выражением вида (e1)(e2), где e1 – первый вектор, e2 – второй. Исходная пара векторов представляет собой матрицу с двумя строками e1 и e2.

Вспомним, что скалярное произведение – это

сумма всех произведений
(d) попарно соответствующих компонент
подлежащих перемножению векторов.

Прочитаем это определение «с конца». Нужно получить, во-первых, попарно компоненты векторов e1 и e2, во-вторых, все произведения этих пар, в-третьих, сумму всех этих произведений.

Итак, план (программа) наших действий состоит из трех последовательных шагов, причем результат предыдущего шага непосредственно используется последующим шагом.

Следовательно, наша программа представляет собой композицию функций

f3 * f2 * f1.

Какие же это функции?

Функция f1 определяет то, что нужно сделать «во-первых». Если даны два вектора, например

(b1) (10 20 30) (3 2 1),

то нужно получить их компоненты попарно:

(b2) (10 3) (20 2) (30 1).

С этим мы уже встречались, так работает функция «транс». Естественно положить f1=транс.

Функция f2 определяет то, что нужно сделать «во-вторых»: получить все произведения пар. В нашем примере это выражение

(b3) 30 40 30

Такое выражение получится, если функцию «умножить» применить к каждому подвыражению выражения (b2). С подобным мы тоже встречались – так работает общая аппликация «А» с аргументом «умножить» (×). Значит, естественно положить f2 =(Ax).

Наконец, f3 определяет, что нужно сделать «в-третьих»: получить общую сумму всех компонент (b3), то есть

(b4) 100

Такое выражение получится, если к (b3) применить форму «редукция» с аргументом «сложить» (+). Значит, естественно положить f3 = (/ +).

Итак, можно выписать нашу программу-формулу полностью:

(/+) * (Ax) * транс.

Эта формула описывает именно ту функцию, которая решает нашу задачу, то есть вычисляет скалярное произведение. Использовать ее можно, как и раньше, двумя способами: либо непосредственно применять к обрабатываемому выражению

:{((/+)*(Ax)*транс) (10 20 30)(3 2 1)} = 100,

либо ввести для нее название, например IP:

IP{e} -> :{((/+)* (Ax) *транс) e}, –

и использовать как обычную МТ-функцию:

IP{(10 20 30) (3 2 1 } = 100.

Как видим, наша программа полностью соответствует определению скалярного произведения – все слова в этом определении использованы, и ничего лишнего не понадобилось вводить (мы записали программу, не использовав ни одного лишнего понятия (!)).

Намеченный идеал концептуальной ясности для данной программы достигнут. Для других программ вопрос открыт, но направление должно чувствоваться. С другой стороны, мы показали, как средства развития в модели МТ, позволяя вводить адекватные понятия (абстракции), помогают бороться со сложностью создания программ.

> **Задача.** Можно ли аналогичные средства ввести в Паскале, Фортране, Бейсике? Дайте обоснованный ответ.

Сравнение функциональной программы с программой на Паскале. Рассмотрим фрагмент программы на Паскале:

```
       c := 0;
(pp)   for i:=l to n do
       c := c + a[i] * b[i];
```

Он вычисляет скалярное произведение двух векторов a и b.

Попытаемся сопоставить его с определением скалярного произведения **(d)**.

Во-первых, сразу видно, что естественная композиция функций в программе (pp) не отражена. Пришлось заменить ее последовательными действиями с компонентами векторов.

Во-вторых, пришлось ввести пять названий c,i,n,a,b, никак не фигурирующих в исходной постановке задачи. Причем если по отношению к a,b и c еще можно сказать, что это обозначения исходных данных и результата, то что такое i и зачем понадобилось n?

Ответ таков, что на Паскале со структурами-массивами по-другому работать нельзя. Мы работали с выражением в модели МТ как с целостным объектом, а в Паскале над массивами возможны лишь «мелкие» поэлементные операции (для этого понадобилась переменная i). К тому же нельзя узнать размер массива, необходимо явно указывать этот размер – (n).

В-третьих, мы уже говорили о возможности распараллелить работу по функциональной программе-формуле. А как это сделать в программе (pp)? Опять сравнение не в пользу Паскаля.

Задача. Найдите аргументы в пользу Паскаля.

Замечание. Программа скалярного произведения в модели Б – это формула, операциями в которой служат формы, а операндами – основные скалярные функции (+, x) и некоторые другие (транс). В этой связи интересно напомнить, что Джон Бэкус – отец Фортрана. Последний тоже начинался как Formula Translation (и «испортился» под натиском «эффективности»). Так что Джон Бэкус пронес идею «формульного» программирования через многие годы, от своего первого знаменитого Фортрана до теперь уже так же знаменитого «функционального стиля». Излагая модель Б, мы воспользуемся лекцией, прочитанной Джоном Бэкусом по случаю вручения ему премии Тьюринга за выдающийся вклад в информатику [22].

2.2. Функциональное программирование в стиле Бэкуса (модель Б)

Мы показали, как функции высших порядков помогают писать концептуально ясные программы. Теперь займемся моделью Б подробнее. Одна из целей – познакомить с разработанной в этой модели алгеброй программ и с ее применением для доказательства эквивалентности программ. Чтобы законы в этой алгебре были относительно простыми, нам понадобится, во-первых, ограничить класс обрабатываемых объектов – считать объектами не произвольные выражения, а только МТ-термы (то есть термы в смысле модели МТ); во-вторых, так подправить определения форм, чтобы их применение всегда давало объекты. В-третьих, придется ввести функции, позволяющие раскрывать и создавать термы.

Для выразительности и краткости при наших определениях будем пользоваться общематематической символикой. Однако все нужные объекты, функции и формы можно без принципиальных трудностей ввести и средствами модели МТ.

2.2.1. Модель Бэкуса с точки зрения концептуальной схемы

Имея опыт работы со структуризованными объектами (выражениями), формами и рекурсивными определениями, который мы приобрели, работая в модели МТ, можно с самого начала рассматривать модель Бэкуса (модель Б) по нашей кон-

цептуальной схеме. Чтобы не загромождать изложение, не будем постоянно подчеркивать различие между знаками в языке Б (модели Б) и их денотатами, надеясь, что из контекста всегда будет ясно, о чем идет речь. Например, будем называть формой как функцию высшего порядка (денотат), так и представляющее ее выражение (знак). Соответственно, примитивной функцией будем называть как ее идентификатор (знак), так и обозначаемое этим идентификатором отображение из объектов в объекты (денотат).

Базис. В модели два скалярных типа – атомы и примитивные функции. Первые служат для конструирования объектов, вторые – для конструирования функций. Объекты и формы – это два структурных типа. Имеется только одна операция – аппликация.

Развитие. Единственным средством развития служит возможность пополнять набор D определений функций. Делается это с помощью фиксированного набора форм и примитивных функций. Определения могут быть рекурсивными.

2.2.2. Объекты

Объект – это либо атом, либо кортеж (последовательность) вида

< X1, ... , Xn >,

где Xi – либо объект, либо специальный знак <?> – «не определено».

Таким образом, выбор фиксированного множества A атомов полностью определяет множество всех объектов O.

Будем считать, что в A входят (то есть служат атомами) идентификаторы, числа и некоторые специальные знаки (T,F и т. п.). Выделен специальный атом <> – это единственный объект, который считается одновременно и атомом, и (пустым) кортежем.

> **Замечание.** Аналогично спискам Лиспа нетрудно представить Б-объекты МТ-выражениями, введя подходящие обозначения для специальных объектов и заключая последовательности объектов в круглые скобки. Это же относится и к последующему неформальному изложению модели Б (хороший источник полезных упражнений по представлению Б-понятий МТ-понятиями).

Все объекты, содержащие <?> в качестве элемента, считаются по определению равными <?> (то есть знаки различны, а денотаты равны). Будем считать, что все такие объекты до применения к ним каких бы то ни было операций заменяются «каноническим» представлением «<?>».

Примеры объектов: <?>, 15, AB3, <AB,1,2,3>, <a,<<B>,C>,D>.

2.2.3. Аппликация

Смысл этой операции известен. Обозначать ее будем по-прежнему через «:», однако использовать не как префиксную, а как инфиксную операцию. Так что если f – функция и X – объект, то

f : X

обозначает результат применения функции f к объекту X. Например:

+ : <1,2> = 3, 1 : <A,B,C> = A, 2 : <A,B,C> = B,
t1 : <A,B,C> = <B,C>,

где слева от знака аппликации «:» выписаны знаки функций сложения, извлечения первого элемента, извлечения второго элемента и хвоста кортежа соответственно.

2.2.4. Функции

Все Б-функции отображают объекты в объекты (то есть имеют тип O -> O) и сохраняют неопределенность (то есть f : <?> = <?> для всех f).

Каждый знак Б-функции – это либо знак примитивной функции, либо знак формы, либо знак функции, определенной в D. Другими словами, в модели Б различаются, с одной стороны, предопределенные функции и формы, а с другой – функции, определяемые программистом с помощью пополнения D.

Равенство f : X = <?> возможно всего в двух случаях, которые полезно различать.

Во-первых, выполнение операции «:» может завершаться и давать в результате <?>.

Во-вторых, оно может оказаться бесконечным – тогда это равенство считается справедливым уже по определению операции «:». Другими словами, виды ненормального выполнения аппликации в модели Б «склеиваются», не различаются.

2.2.5. Условные выражения Маккарти

Ниже будем пользоваться модификацией так называемых условных выражений Маккарти. **Условное выражение Маккарти** – это запись вида

(P1 --> E1, ... ,Pn --> En, T --> E),

где через P с возможными индексами обозначены условия (предикаты, логические функции), а через E с возможными индексами – выражения, вычисляющие произвольные объекты. При конкретной интерпретации входящих в эту запись переменных ее значение получается по следующему правилу. Последовательно вычисляются условия, начиная с первого, пока не найдется истинное (**такое всегда найдется. Почему?**). Затем вычисляется соответствующее выражение, значение которого и становится значением всего условного выражения Маккарти.

Аналоги такой конструкции широко используются в ЯП. В сущности, возможность согласования с левой частью конкретного МТ-предложения можно рассматривать как аналог условия Pi, а правую часть МТ-предложения – как аналог выражения Ei.

Упражнение. Укажите различия между Pi, Ei и названными их МТ-аналогами.

Вслед за Джоном Бэкусом будем записывать условные выражения Маккарти в виде

P1 --> E1;...; Pn --> En; E,

то есть опуская внешние скобки и последнее тождественно истинное условие, а также используя точку с запятой в качестве разделителя (запятая занята под знак формы «конструкция»).

2.2.6. Примеры примитивных функций

Некоторые из уже известных вам функций опишем заново, во-первых, чтобы можно было освоиться с введенными обозначениями, и, во-вторых, потому, что они играют важную роль в **алгебре программ (АП)**, к описанию и применению которой мы стремимся. Определения некоторых ранее рассмотренных функций уточнены здесь с тем, чтобы упростить АП (соответствующая модификация МТ-определений может служить полезным упражнением). Отличия касаются, как правило, тонкостей (действий с пустыми и неопределенными объектами, а также учета внешних скобок в представлении объектов), однако эти тонкости существенны с точки зрения АП.

Селекторы (селекторные функции). Будем использовать целые числа для обозначения функций, выбирающих из кортежей-объектов элементы с соответствующим номером:

1 : X :: X = <X1,...,Xn> --> X1; <?>,

то есть функция определяется через аппликацию (этот прием мы уже применяли в модели МТ). Знак «::» используется, как и раньше, в смысле «есть по определению», чтобы отличать определение от обычного равенства. Приведенная запись означает, что применение функции 1 к объекту X дает результат X1 (первый элемент кортежа), если X – кортеж, иначе – не определено (то есть аппликация завершает вычисление с результатом <?>).

Вообще, для положительного целого s:

s : X :: X = <X1,...,Xn> & n>=s --> Xs; <?>.

Здесь в условном выражении Маккарти употреблено более сложное условие вида

X = <X1, ... ,Xn> & n > = s.

Замечание. Такого рода определения (и это, и предыдущее) следует рассматривать лишь как достаточно понятное сокращение точного определения, которое может быть дано, например, в модели МТ. Модель МТ неплохо подходит для таких определений, потому что в ее базисе имеются мощные средства анализа и синтеза. Ведь нужно сказать следующее: если объект X имеет вид

<X1,...,Xn>

и длина кортежа больше s, то заменить X на s-ю компоненту кортежа. Если же X не имеет такого вида («согласование невозможно» с соответствующим МТ-предложением), то выдать <?>. Например:

5{(t1 t2 t3 t4 t5 e)}---> t5 .
5{e} ---> <?> .

Хвост

t1 : X :: X = <X1> --> <>;
X = <X1,...,Xn> & n > = 2 -->
<X2, ... ,Xn>;
<?>.

Отличие от МТ-функции «хвост» в том, что результат всегда в скобках, если определен и непуст.

Тождественная функция

id : X :: X

Логические функции – атом, равенство, пустой

атом : X :: (X – это атом) --> T;
X /= <?> --> F; <?>.

Таким образом, атом:<?> = <?> (сохраняет неопределенность),

eq : X :: X = <Y,Z> & Y = Z --> T;
X = <Y,Z> & Y /= Z --> F; <?>.

Например, eq:<2,2,3> = <?>, так как аргумент не пара, а тройка.

null : X :: X = <> --> T; X /= <?> --> F; <?>.

Сложение, вычитание, умножение, деление

+ : X :: X = <Y,Z> & числа (Y,Z) --> Y+Z; <?>.
- : X :: аналогично.
Mult : X :: аналогично. [Привычная звездочка занята]
div : X :: аналогично. [Косая черта занята]

Расписать левым, расписать правым

distl : X :: X = <Y,<>> --> <>;
X = <Y,<Z1,...,Zn>> --> <<Y,Z1>,...,<Y,Zn>>;
<?>.
distr : X :: X = <<>,Y> --> <>;
X = <<Z1,...,Zn>,Y> --> <<Z1,Y>,...,<Zn,Y>>; <?>.

Функция distl присоединяет левый элемент аргумента ко всем элементам кортежа, который служит правым элементом аргумента функции, а функция distr, наоборот, присоединяет правый элемент аргумента. Например:

distl:<<A,B>,<C,D>> = <<<A,B>,C>,<<A,B>,D>>.
distr:<<A,B>,<C,D>> = <<A,<C,D>>,<B,<C,D>>>.

Транспонирование

```
trans : X ::
      X =<<X11,...,X1m>,    <<X11,...,Xn1>,
         <X21,...,X2m>, --> <X12,...,Xn2>,
                ...                   ...
         <Xn1,...,Xnm>>     <X1m,...,Xnm>>;
<?>.
```

Здесь для наглядности строки матрицы выписаны друг под другом. С подобной функцией мы работали в модели МТ. К объектам, не являющимся матрицами, она неприменима (то есть результат – <?>).

Присоединить
appendl : X :: X = <Y,<>> --> <Y>;
X = <Y,<Z1,...,Zn>> --> <Y,Z1,...,Zn>;
<?>.

Таким образом, эта функция присоединяет левый элемент аргумента в качестве самого левого элемента правой компоненты аргумента. Поэтому ее естественно назвать «присоединить левый». Например:

append1:<<A,B>,<C,D>> = <<A,B>,C,D>.
appendr : X :: X = <<>,Y> --> <Y>;
X =<<Y1, ... ,Yn,Z> --> <Y1, ... ,Yn,Z>;
<?>.

Такую функцию естественно назвать «присоединить правый».
Например:

appendr : <<A,B>,<C,D>> = <A,B,<C,D>>.

2.2.7. Примеры форм, частично известных по работе в модели МТ

Композиция (обозначение – *)

(f *g) :X::f : (g :X).

Конструкция (обозначение – ,)

(f1, ... ,fn) : X :: <f1:X, ... , fn:X>.

Замечание. В отличие от МТ-конструкции, здесь результат – целостный объект в скобках, причем все n результатов применения каждой функции – также целостные объекты (то есть каждая компонента общего результата явно отделена от остальных). В МТ-конструкции отделить результаты функций-компонент в общем случае невозможно, так как это могут быть не термы, а выражения, не разделенные между собой явно. По указанной причине для МТ-конструкции не выполняются некоторые важные алгебраические законы, верные в случае Б-конструкции. Так что указанные отличия МТ- и Б-конструкций существенны.

Упражнение. Определите МТ-конструкцию со свойствами, аналогичными Б-конструкции.

Подсказка. Нужно использовать не конкатенацию выражений, а функцию, аналогичную созданию списка из атомов и (или) списков.

Вопрос. Какие важные алгебраические законы не выполняются для МТ-конструкции?

Подсказка. Что будет, если некоторый объект присоединить левым, а затем выбрать первый (то есть левый) элемент? Зависит ли результат от структуры этого объекта в случае Б-конструкции? А в случае МТ-конструкции?

Условие

```
(p --> f;g) : X :: (p:X) = T --> f:X;
           (p:X) = F --> g:X; <?>.
```

В отличие от условного выражения Маккарти, в форме «условие» все аргументы – функции, а не объекты. Вслед за Бэкусом вместо

```
(p1 --> f1; (p2 --> f2;g))
```

будем писать без скобок:

```
p1 --> f1; p2 --> f2; g.
```

Генератор постоянных

```
const(X) : Y :: Y – <?> --> <?>; X.
```

Аргумент этой формы – некоторый объект, а результат – функция, значение которой на всех определенных объектах совпадает с объектом, заданным в качестве аргумента формы. Другими словами, результатом формы служит функция-константа с заданным значением. Например, для любого Y /= <?>

```
const (<A,B>):Y = <A,B>.
```

Редукция (обозначение – /)

```
/f : X :: X=<X1>--> X1;
X = <X1,...,Xn>&n >= 2 -->
f:<Xl,/f:<X2,...,Xn>>;
<?>.
```

Общая аппликация (обозначение – A)

```
Af: X :: X = <> --> <>;
X= <X1, ... ,Xn> --> <f:Xl,    ,f:Xn>;
<?>.
```

Отличия от МТ-общей-аппликации аналогичны отличиям МТ- и Б-конструкций.

Упражнение. Написать определение МТ-общей аппликации, соответствующее приведенному Б-определению.

Итерация

```
(while p f) : X ::
      (p:X) =T --> (while p f):(f:X);
      (p:X) = F --> X; <?>.
```

Смысл этой формы – в том, что функция f в общем случае многократно применяется к объекту X до тех пор, пока не станет ложным результат применения к X логической функции (условию) p.

Специализатор (обозначение – s)

```
(s f X) : Y :: f:<X,Y>.
```

У этой формы два аргумента: бинарная операция f (функция от двухэлементных объектов) и объект X. Результат формы – унарная операция, получающаяся из f при ее специализации (иногда говорят «конкретизации») за счет отождествления первого операнда с объектом X. Например:

(s + 1) : Y =+:<1,Y>,

то есть 1 + Y в обычных инфиксных обозначениях.

2.2.8. Определения

Б-определение новой функции – это выражение вида

DEF I :: r,

где в качестве I указано не использованное ранее название функции (функциональный символ), а в качестве r – функциональная форма (которая может зависеть от I; допустимы и рекурсивные определения). Например:

DEF last :: null * t1 --> 1; last * t1,

где справа от знака «::» – форма «условие», в которую входят в качестве p композиция null * t1, в качестве f – селекторная функция «1», а в качестве g – композиция last * t1.

Так что last:<A,B>= B. **Убедитесь, что это действительно верно!**

Нельзя выбирать названия новых функций так, чтобы они совпадали с уже введенными или предопределенными. Использование в D функционального символа всюду вне левой части определения (то есть в качестве I) формально означает, что вместо него нужно подставить соответствующее r и попытаться вычислить полученную аппликацию. Если в процессе вычисления снова встретится функциональный символ, определенный в D, то снова заменить его соответствующей правой частью определения и т. д.

Ясно, что можно зациклиться. Как уже было сказано, это один из способов получить <?> в качестве результата. Конечно, в разумных рекурсивных определениях прямо или косвенно применяются условные выражения Маккарти, которые позволяют вычислить аппликацию за конечное число шагов (**за счет чего?**).

Б-определение – в сущности, еще одна форма, ставящая в соответствие функциям-аргументам определяемую функцию.

Тем самым мы закончили описывать базис модели Б, а также основное (и единственное) средство развития – форму DEF.

2.2.9. Программа вычисления факториала

Продемонстрируем сказанное на примере рекурсивной программы, по-прежнему стремясь к идеалу концептуальной ясности.

Рассмотрим задачу вычисления факториала. Конечно, наша цель – не просто запрограммировать факториал, а показать, как это делается с помощью форм. Начнем, как и раньше, с математического определения нужной функции «факториал» (обозначение – !) и снова применим пошаговую детализацию.

Обычное математическое определение:

!n равен 1, если n = 0; иначе равен n, умноженному на !(n−1).

Как такое определение переписать в стиле Бэкуса?

На первом шаге нужно выявить функции, на которые непосредственно разлагается исходная функция «факториал». Ее исходное определение не устраивает нас потому, что оно не прямо разлагает факториал на компоненты-функции, а описывает его (факториала) результат через результаты применения некоторых функций.

С учетом сказанного вернемся к исходному определению факториала. Что известно про функцию «!»? То, что она разлагается на различные составные части в зависимости от некоторого свойства аргумента (в зависимости от его равенства нулю).

Следовательно, можно представить факториал условной формой

DEF ! :: (p --> f;g),

где p,f и g – пока не определенные функции.

Вот и выполнен первый шаг детализации. Теперь займемся введенными функциями.

Что известно про p? Это функция, проверяющая равенство нулю исходного аргумента. Итак, p можно представить в виде композиции функции **eq** и некоторой пока еще не определенной функции, которая готовит для eq аргументы (точнее, формально один аргумент-кортеж из двух содержательных аргументов). Получим

DEF p :: eq * f1.

Что делает f1? Она ведь должна приготовить аргументы для проверки исходного аргумента на равенство другому объекту, а именно нулю. Другими словами, ее результатом должна быть пара, составленная из исходного аргумента факториала и нуля. Пару естественно получить формой «конструкция». Так что

DEF f1 :: f2 , f3.

Что делает f2? Поставляет первый аргумент пары. Но ведь это исходный аргумент без изменений! Следовательно,

DEF f2 :: id.

А что делает f3? Поставляет второй аргумент пары. Но ведь это нуль! Отлично, значит, f3 – постоянная, которую естественно определить через форму const:

DEF f3 :: const (0).

Итак, функция p определена полностью. Продолжим детализацию для f и g.

Что делает f? Всегда дает в результате единицу. Значит, это постоянная, которую также легко выразить через const.

DEF f :: const(1).

А что делает g? Вычисляет произведение двух объектов, каждый из которых, как теперь уже нетрудно понять, должен доставляться своей функцией. Значит, g естественно представить композицией функций «mult» (умножить) и некоторой конструкции двух функций:

DEF g :: mult * (g1 , g2).

Очевидно, что g1 совпадает с id (**почему?**). А g2 представляет собой композицию определяемой функции «!» и функции g3, вычитающей единицу из исходного аргумента. Поэтому

DEF g2 :: ! * g3,

где g3, в свою очередь, представляется как композиция вычитания и конструкции, готовящей для этого вычитания аргумент. Позволим себе пропустить шаг подробной детализации и сразу написать

DEF g3 :: – * (id , const(1)).

Пошаговая детализация полностью завершена. Программа в модели Б, вычисляющая факториал, написана. Метод пошаговой детализации продемонстрирован. Но, конечно, программа не обязана быть такой длинной, и не все шаги обязательны. Полученные на промежуточных шагах определения можно убрать, подставив соответствующие формулы вместо обозначений функций. В нашем случае получим следующее функциональное соотношение, определяющее факториал:

DEF ! :: eq0 --> const(1); mult*(id, !*sub1),

где через eq0 переобозначена для наглядности функция p, а через sub1 – функция g3.

Кстати, eq0= (s eq 0). Верно ли, что sub1= (s – 1)?

Замечательное свойство пошаговой детализации (функциональной декомпозиции) в модели Б состоит в том, что нам ни разу не понадобилось исправлять определения ранее введенных функций. Другими словами, функциональная декомпозиция не зависит от контекста (она, как говорят, **контекстно свободна**). Это важнейшее преимущество функционального программирования с точки зрения борьбы со сложностью программ.

Вопрос. За счет чего оно достигнуто?

Упражнение. Запрограммируйте в стиле Бэкуса другое определение факториала: !n равен произведению всех различных натуральных чисел, меньших или равных n.

2.2.10. Программа перемножения матриц

Напишем в стиле Бэкуса программу, перемножающую две прямоугольные матрицы (согласованных размеров). Снова применим метод пошаговой детализации. Начнем, как обычно, с постановки задачи, то есть с определения функции MM (matrix multiply), которую предстоит запрограммировать.

Результат умножения двух матриц B1(m,n) и B2(n,k) –

(def) **это такая матрица C(m,k), каждый элемент c(i,j) которой – скалярное произведение i-й строки матрицы B1 на j-й столбец матрицы B2**.

До сих пор в наших примерах мы имели дело с программами, работающими с относительно просто устроенными данными. Поэтому можно было не заниматься проблемой представления данных специально. Между тем в программистском фольклоре бытует крылатая фраза, довольно точно отражающая трудоемкость и значимость отдельных аспектов программирования: «**Дайте мне структуру данных, а уж программу я и сам напишу!**» Мы еще не раз поговорим о данных, а пока займемся представлением исходных данных в нашей задаче, то есть представим матрицы в виде Б-объектов (других типов данных в модели Б просто нет (!)).

При описании представления нам потребуется говорить о длине объектов-кортежей. Поэтому введем еще одну примитивную функцию

leng : X :: X =<>--> 0;
X=<X1, ... ,Xn> --> n; <?>.

Функция leng вычисляет число непосредственных компонент объекта X (его длину как кортежа). Например:

leng:<A,<B,C>,D> = 3,
leng:(2:<A,<B,C>,D>) = 2.

Аргумент X функции MM будет парой (объектом длины 2), компоненты которой представляют исходные матрицы. Так что

1:X=B1 , 2:X=B2

(здесь B1 и B2 используются как обозначения матриц, а не как Б-атомы!). При этом матрицы, в свою очередь, будут представлены кортежами строк. Например, если

B1 = 3 5 7,   B2 = 9 5
     2 4 6   6 3
                    1 2,

то

X =<<<3,5,7>,<2,4,6>>,<<9,5>,<6,3>,<1,2>>>.

Так что

leng:(l:X) = 2, и это число строк в матрице B1,
leng:(l:(l:X)) = 3 (число столбцов в B1),
leng:(2:X) =3 (число строк в матрице B2),
leng:(l:(2:X)) = 2 (число столбцов в B2).

Итак, мы научились представлять матрицы «по строкам» и говорить об их размерах в терминах «объектного представления». Осталось отметить, что если объект Y представляет матрицу, то ее элемент, стоящий на пересечении i-й строки и j-го столбца, можно получить с помощью функции

j * i.

При этом элементы матрицы B1 можно получить из X функцией

j*i* 1,

а элементы матрицы B2 – функцией

j*i*2.

Теперь мы готовы приступить к первому шагу функциональной декомпозиции.

Первый шаг. Как и раньше, начнем с «конца» определения (def). Там сказано, что «каждый элемент» результата – это скалярное произведение некоторых объектов. Следовательно, ММ можно определить как композицию двух функций, из которых внутренняя готовит сомножители для скалярных произведений, а внешняя выполняет умножение для всех заготовленных аргументов.

DEF MM :: f2 * f1.

Можно ли сразу же продолжить функциональную декомпозицию? На первый взгляд можно, тем более что совсем недавно шла речь о ее независимости от контекста.

Но как же детализировать, например, f2, не зная, какова структура аргумента у этой функции? Все дело в том, что мы по существу еще не завершили предыдущего шага декомпозиции – мало обозначить компоненты функциональной формы, нужно еще описать их внешний эффект (или, как говорят, спроектировать их внешнюю спецификацию). Другими словами, нужно решить, каково у каждой из этих функций множество исходных данных, каково множество допустимых результатов и какое именно отображение из первого множества во второе каждая из функций осуществляет.

В нашем случае для ММ все это ясно (и тем самым определены исходные данные для f1 и результаты для f2). Нужно «разделить сферы влияния» f1 и f2, выбрав представление для аргумента функции f2 (и тем самым для результата функции f1).

Вспомнив (def), нетрудно понять, что строение аргумента функции f2 должно отличаться от строения результата лишь тем, что на местах элементов матрицы С должны находиться те самые пары объектов-кортежей, из которых соответствующие c(i,j) будут получены скалярным умножением. Например, в случае наших B1 и B2 аргумент функции f2 должен представлять собой объект, соответствующий матрице размером 2 на 2, из которого функция, например, 2 * 1 извлекает пару <<3,5,7>,<5,3,2>>. Назовем поэтому аргумент функции f2 **матрицей пар**. Вот теперь можно продолжить декомпозицию.

На втором шаге займемся функцией f2. Она должна применять функцию IP (скалярное умножение) КО ВСЕМ элементам матрицы пар. Если бы в нашем распоряжении была форма, которая применяет свой аргумент-функцию ко всем элементам матрицы (то есть кортежа кортежей), то было бы ясно, как представить f2 через эту форму и IP (**кстати, как это сделать?**). Но у нас есть лишь общая аппликация А (применяющая заданную ей в качестве аргумента функцию ко всем элементам кортежа). Таким образом, если ввести определение

DEF f2 :: (A f3),

то f2 окажется представленной через общую аппликацию с функцией f3, применяемой ко всем «строкам» матрицы пар. Осталось обеспечить, чтобы при каждом своем применении f3 применяла IP ко всем элементам «строки» (то есть кортежа пар). Ясно, что нужная функция легко выражается через А и IP:

DEF f3 :: (A IP).

Подставляя вместо f3 ее определение и вслед за Бэкусом убирая вторые скобки, завершаем декомпозицию f2 определением

DEF f2 :: (AA IP).

Вполне можно считать, что через AA обозначена та самая форма, которая применяет свой аргумент ко всем элементам кортежа кортежей. Ее называют **двойной общей аппликацией**.

На третьем шаге детализации займемся f1. Как мы выяснили, эта функция из пары исходных матриц получает матрицу пар. При этом элемент матрицы пар составлен из i-й строки первой исходной матрицы и j-го столбца второй матрицы. Другими словами, каждая строка первой матрицы сочетается с каждым столбцом второй. Например, в случае матриц B1 и B2 функция 2*1 должна выбрать из матрицы пар объект <<3,5,7>,<2,4,6>>. Но раз каждая сочетается с каждым, естественно возникает идея получить матрицу пар «расписывающими» функциями – distl и distr. Однако, чтобы «расписать», нужно иметь, что «расписывать». И если строки первой матрицы представлены объектами-кортежами, то объектов, представляющих столбцы второй матрицы, у нас нет – ведь матрицы представлены «по строкам»! Поэтому следует представить f1 композицией функций, внутренняя из которых располагает вторую матрицу «по столбцам», внешняя – «расписывает» матрицу пар. Итак, третий шаг детализации завершает определение

DEF f1 :: f5 * f4.

При этом считаем, что внешняя спецификация новых функций очевидна; кстати, понимаете ли вы, что у них на входе и что на выходе?

На четвертом шаге функциональной декомпозиции займемся f4. Содержательно ее назначение – подготовить из исходной новую пару матриц, сохранив ее первую компоненту и переписав «по столбцам» вторую. При уже имеющемся у нас опыте Б-программирования можно сразу выписать определение

DEF f4 :: (1 , trans * 2).

Действительно, такая конструкция составляет новую пару из первой компоненты исходной пары и транспонированной второй.

На пятом шаге рассмотрим f5. Ее назначение – по двум матрицам получить матрицу пар, сочетая каждую строку первой матрицы с каждой строкой второй. При этом i-я строка матрицы пар (то есть ее i-й элемент как кортежа) представляет собой кортеж, полученный сочетанием i-й строки первой матрицы с каждой строкой второй матрицы в естественном порядке. Значит, если удастся сначала составить кортеж пар, в которых первым элементом будет i-я строка первой матрицы, а вторым – вся вторая матрица целиком, то затем можно каждую такую пару <строка,матрица> превратить в кортеж пар <строка,строка>.

Здесь пригодятся наши «расписывающие» функции. Действительно, кортеж пар <строка,матрица> получается применением к паре матриц функции distr («расписать правым» – ведь правая компонента сочетается со всеми компонентами-строками левой матрицы), а затем из каждой такой пары можно получить кортеж вида <строка,строка> применением общей аппликации с функцией distl

(ведь левая компонента ее аргумента сочетается со всеми компонентами правой матрицы). Итак, декомпозицию f5 завершает определение

DEF f5 :: (A distl) * distr.

Тем самым оказывается полностью завершенным и процесс пошаговой функциональной декомпозиции функции MM. Подставляя вместо обозначений промежуточных функций их определения, получаем **определение MM**

```
DEF MM :: (AA IP)*(A distl)*distr*(l,trans*2).
           *****     ---------------  ========
```

Таким образом, MM разлагается на подготовку соответствия строк первой матрицы столбцам второй (выделено двойной чертой), подготовку пар (выделено одинарной чертой) и собственно перемножение (выделено звездочками).

И опять **ничего лишнего**, ***структура программы-формулы получена непосредственно из постановки задачи***! Принцип концептуальной ясности снова действует. Конечно, сначала такая программа может показаться и не слишком понятной. Новое, тем более принципиально новое, усваивается не сразу. Однако излагаемый стиль программирования стоит затрачиваемых на его освоение усилий! Обратите внимание – нет имен для промежуточных данных, нет переменных, нет особых управляющих конструктов, нет процедур, нет инициализации (установки начальных значений).

Как видите, многого нет с точки зрения традиционного программирования в стиле фон Неймана. Зато есть **концептуальная ясность**, есть **обобщенность** (в качестве исходных данных пригодны любые согласованные по размерам матрицы; **кстати, где учтено требование согласованности размеров?**), есть возможность **распараллеливания**, есть, наконец, возможность **оптимизации** (так как вычисление скалярных произведений независимо, их можно вычислять и последовательно, не требуя большой памяти, – эту возможность мы еще продемонстрируем).

> **Замечание.** Важно понять, что стиль программирования, ориентированный на концептуальную ясность, предполагает концентрацию внимания исключительно на сути решаемой задачи при как можно более полном игнорировании таких второстепенных на этом этапе вопросов, как ресурсоемкость решения с точки зрения исполнителя. Самое главное – найти правильное и понятное (убедительно правильное, концептуально ясное) решение.
>
> Вот когда есть правильное решение, и оно не устраивает по ресурсоемкости, осмысленно тратить время и силы на постановку и решение новой задачи – искать лучшее решение. (Кстати, это тоже пошаговая детализация – детализация поиска оптимального решения.) Выделить первый шаг такой детализации (на котором в сущности конструктивно доказывается теорема существования решения) принципиально важно с методологической и технологической точек зрения. Ведь на последующих шагах оптимальное решение ищется в надежных и комфортных условиях – есть куда отступать и есть с чем сравнивать.

Такой подход требует определенной психологической подготовки. Например, когда в процессе пошаговой детализации мы обнаружили, что i-я строка понадобится для IP много раз, то с точки зрения концептуальной ясности вывод очеви-

ден – размножить каждую строку в нужном количестве экземпляров! При другом стиле программирования такое даже в голову не придет «квалифицированному» программисту – он интуитивно игнорирует его как расточительное по ресурсам. Но именно такое решение ведет не только к ясности, но и к эффективности, когда память недорога, а процессоров сколько угодно. И, во всяком случае, оно открывает путь к анализу имеющегося правильного решения вместо нерационального расхода человеческих ресурсов на, возможно, ненужную оптимизацию или попытку воспользоваться неправильной, но показавшейся привлекательной программой.

Итак, мы рассмотрели три модели ЯП (в основном с технологической позиции, то есть с точки зрения программиста).

Модель Н – самая близкая к ЯП, получившим наибольшее распространение (неймановским ЯП). Вместе с тем она проявляет основной источник сложности программирования на этих ЯП – неразработанность средств функциональной декомпозиции.

Модель МТ демонстрирует один из перспективных подходов к созданию таких средств – она предоставляет мощные средства развития, основанные на выделении двух ключевых абстракций (анализа и синтеза текстов посредством образцов).

Модель Б указывает путь к рациональному программированию, ключевые идеи которого опираются на многовековой опыт применения алгебраических формул. С другой стороны, пока эта модель из рассмотренных нами самая далекая от современного массового программирования (однако она находится в центре внимания создателей перспективных языков и машин).

Таким образом, ЯП можно строить на весьма различных базисах и различных средствах развития.

Глава 3

Доказательное программирование (модель Д)

3.1. Зачем оно нужно 316
3.2. Доказательное программирование методом Бэкуса 316
3.3. Доказательное программирование методом Хоара 321

3.1. Зачем оно нужно

Если согласиться, что надежность – важнейшее качество программы, то понятен интерес современных исследователей к таким методам создания программ, которые бы в максимальной степени способствовали устранению ошибок в программах. Среди них особое место принадлежит классу методов, в основе которых лежит концепция математического доказательства правильности программы (или, короче, **доказательного программирования**).

Конечно, такая концепция должна подразумевать представление как самой программы, так и требований к ней определенными математическими объектами. Другими словами, в каждом методе доказательного программирования строится некоторая математическая модель программы и внешнего мира (программной среды), в котором выразимо понятие правильности программ.

Ясно, что доказательное программирование в принципе не в состоянии полностью избавить от содержательных ошибок в программе уже хотя бы потому, что указанные математические модели (в том числе математическая модель ошибки) – лишь приближение к реальному внешнему миру программы.

Вместе с тем доказательное программирование в целом получило столь серьезное развитие, что некоторые специалисты даже склонны считать владение им обязательным атрибутом квалифицированного программиста. Если оно и не избавляет от ошибок, то по крайней мере может способствовать весьма тщательному, хорошо структурированному проектированию и анализу программы. Поскольку при этом ведется работа с точными математическими объектами, существенная часть ее может быть автоматизирована.

В задачу книги не входит подробное изложение методов доказательного программирования. Нет нужды конкурировать с прекрасным учебником такого признанного мастера, как Дэвид Грис [23]. Нас интересуют прежде всего свойства ЯП, полезные для доказательного программирования. Рассмотрим эти свойства на примере двух методов, первый из которых (метод Бэкуса) до сих пор не был освещен в отечественной литературе, а второй (метод Хоара) считается классическим для этой области.

3.2. Доказательное программирование методом Бэкуса

Математическая модель программы в модели Б построена – программа представлена функциональным выражением (формулой). Остается построить модель внешнего мира. Он представлен алгеброй программ. С точки зрения доказательного программирования самое важное, что можно делать с программой в этом мире, – ее можно преобразовывать по законам этой алгебры.

3.2.1. Алгебра программ в модели Б

Наша ближайшая цель – показать, как простота (свобода от контекста) функциональных форм способствует построению системы регулярных преобразований (алгебры программ), сохраняющих смысл программы. В этой алгебре носитель (область значений переменных):

- это класс функций в модели Б, а операции – формы в модели Б. Например:

  (f * g) * h;

- это выражение (формула) в алгебре программ. Результат «вычисления» этого выражения различен в зависимости от значений переменных f,g,h – при конкретной интерпретации переменных это вполне определенная функция. Например, при интерпретации

  (f -> id,g -> id,h -> id)

  получаем

  (f * g) * h = id.

В этой алгебре с программами-выражениями можно обращаться аналогично тому, как в школьной алгебре обращаются с формулами, уравнениями, неравенствами. Применять это умение можно для упрощения программ и доказательства их эквивалентности (а значит, и корректности, если правильность эквивалентной программы установлена или считается очевидной).

Основой для преобразований служит серия законов. Некоторые из них мы перечислим. Например:

(f,g) * h =((f * h),(g * h)).

К этим законам можно относиться либо как к аксиомам, либо как к теоремам (скажем, когда рассматривается конкретная реализация форм в модели МТ). Будем относиться к перечисленным ниже законам как к аксиомам. Лишь для одного из них в качестве примера приведем доказательство.

Законы алгебры программ

1. (f1,...,fn) * g= (f1 * g,...,fn * g).
2. Af * (g1,...,gn) = (f * g1,...,f * gn).
   Здесь «A» обозначает общую аппликацию.
3. /f * (g1,...,gn) = f * (g1,/f * (g2,... ,gn)) для n >= 2.
   /f * <g> = g.
4. (f1 * 1, ... ,fn * n) * (g1, ... ,gn) = (f1 * g1,...,fn * gn).
5. appendl * (f * g, Af * h) = Af * appendl * (g,h).
6. pair & not * null * 1 ->-> appendl*((l * 1,2) , distr*(t1*1,2)) = distr.
7. A(f * g) = Af * Ag .

Теорема. pair & not * null * 1 ->->
appendl*((l * 1,2), distr*(t1*1,2)) = distr .

Другими словами, равенство, которое написано справа от знака «->->», выполнено для класса объектов, удовлетворяющих условию, выписанному слева от «->->» (то есть для пар с непустой первой компонентой).

Доказательство. Идея: слева и справа от знака равенства получается тот же результат для любого объекта из выделяемого условием класса.

Случай 1. x – атом или <?>.

distr : (x,y) = <?>.   (см. опр distr)

t1 * 1 : (x,y) = <?>.   (опр t1).

И так как все функции сохраняют <?>, получаем утверждение теоремы.

Случай 2. x = <x1,...,xn> то есть x – кортеж.

Тогда

appendl * ((1*1,2) , distr * (t1*1,2)) : <x,y> =

    = appendl:<<1:x,y>, distr:<t1:x,y>> =

Если t1:x = <> [<> **обозначает «пусто»**], то **=**

appendl:<<x1,y>,<>> = <<x1,y>> = distr:<x,y>.

Если t1:x /= <>, то =

appendl:<<xl,y>,<<x2,y>,...,<xn,y>>> = distr:<x,y>.

Что и требовалось доказать.

3.2.2. Эквивалентность двух программ перемножения матриц

Наша ближайшая цель – продемонстрировать применение алгебры программ для доказательства эквивалентности двух нетривиальных программ. Одна из них – программа MM на стр. 308, при создании которой мы совершенно игнорировали проблему эффективности (в частности, расхода памяти исполнителя). Вторая – программа MMR, которую мы создадим для решения той же задачи (перемножения матриц), но позаботимся теперь о рациональном использовании памяти исполнителя. Естественно, MMR получится более запутанной. Уверенность в ее правильности (корректности) будет, конечно, более высокой, если удастся доказать ее эквивалентность MM, при создании которой именно правильность (концептуальная ясность) и была основной целью.

Естественно считать MM точной формальной спецификацией задачи перемножения матриц (то есть определением функции «перемножение матриц»). Тогда доказательство эквивалентности некоторой программы, предназначенной для решения той же задачи, программе MM становится доказательством корректности такой программы по отношению к спецификации (доказательством того, что эта программа решает именно поставленную задачу, а не какую-нибудь другую).

Доказательство корректности программ по отношению к их (формальной) спецификации часто называют **верификацией программ**. Таким образом, мы продемонстрируем, в частности, применение алгебры программ для их верификации.

> **Замечание.** Применяемый нами метод верификации принадлежит Бэкусу и не является общепринятым. Чаще, говоря о верификации, имеют в виду спецификации,

написанные на логическом языке, а не на функциональном. Мы поговорим и о таких спецификациях, когда займемся так называемой дедуктивной семантикой программ. Заметим, что все последнее время мы занимались по сути денотационной семантикой – ведь программа в модели Б явно представляет свой денотат (реализуемую функцию) в виде комбинации функций.

Экономная программа R перемножения матриц

Напомним строение программы MM:

```
DEF MM :: AA IP * A distl * distr * (1 , trans * 2).
```

Замечание. Всюду ниже для уменьшения числа скобок будем считать, что операции «A» и «/» имеют высший приоритет по сравнению с «*» и «,». При реализации в модели МТ этого легко добиться, поместив определения первых операций НИЖЕ в поле определений **(почему?)**.

Рассмотрим программу MM', такую, что

```
DEF MM' :: AA IP * A distl * distr.
```

Она «заканчивает» работу MM, начиная уже с пары матриц с транспонированной второй компонентой. Будем оптимизировать именно MM', так как именно в ней находится источник неэффективности, с которым мы намерены бороться. Дело в том, что функция AA IP применима только к матрице пар, которая требует памяти объемом mb + ka, где a и b – объем соответственно первой и второй матриц, k – число столбцов второй матрицы, m – число строк первой. Хотелось бы расходовать память экономнее.

Постараемся определить другую программу (назовем ее R) так, чтобы она реализовала ту же функцию, что и MM', но экономнее расходовала память. Основная идея состоит в том, чтобы перемножение матриц представить как последовательное перемножение очередной строки первой матрицы на вторую матрицу. Тогда можно перемножать последующие строки *на месте, освободившемся после завершения перемножения предыдущих строк*, так что потребуется лишь объем памяти, сравнимый с размером перемножаемых матриц.

Определим вначале программу mM, выполняющую перемножение первой строки первой матрицы на вторую матрицу:

```
DEF mM :: A IP * distl * (1 * 1,2).
```

Действительно, программа mM сначала «расписывает» строки второй матрицы первой строкой первой матрицы (функция distl), а затем скалярно перемножает получившиеся пары строк, что и требовалось.

Теперь определим программу R:

```
DEF R :: null * 1 --> const (<>);
appendl * (mM, MM' * (tl * 1,2)).
```

Таким образом, если первая матрица непустая, то результат функции R получается соединением в один объект (с помощью appendl) результата функции mM (она перемножает первую строку первой матрицы на вторую матрицу) и результата функции MM', которая «хвост» первой матрицы перемножает на вторую матрицу.

Заметим, что если удастся доказать эквивалентность MM' и R, то MM' можно заменить на R и в определении самой R. Так что определение R через MM' можно считать техническим приемом, облегчающим доказательство (не придется заниматься рекурсией). Перепишем определение R без MM':

```
DEF R :: null * 1 --> const(<>);
appendl * (mM, R * (t1 * 1, 2)).
```

Независимо от того, удастся ли доказать эквивалентность R и MM', ясно, что в новом определении R отсутствует двойная общая аппликация, и если вычислять R разумно (как подсказывает внешняя конструкция, то есть сначала вычислить левый ее операнд, а затем правый), то последовательные строки матрицы-результата можно вычислять на одном и том же рабочем пространстве. Этого нам и хотелось!

Итак, сосредоточившись на сути задачи, мы выписали ее спецификацию-функцию, то есть программу MM', а концентрируясь на экономии памяти, получили оптимизированный вариант программы, то есть R. Займемся верификацией программы R.

Верификация программы R. Покажем, что

R = MM'

для всех аргументов, которые нас интересуют, то есть для всех пар. Есть различные способы такого доказательства. «Изюминка» модели Бэкуса состоит в том, что для доказательства свойств программ и, в частности, их верификации можно пользоваться общими алгебраическими законами (соотношениями), справедливыми в этой модели, причем для установления и применения этих законов не нужно вводить никакого нового аппарата – все, что нужно, уже определено в модели Б.

Докажем, что верна следующая

Теорема. pair ->-> MM' = R.

Доказательство

Случай 1: pair & null * 1 ->-> MM' = R.

pair & (null * 1) ->-> R = const(<>).

По определению R

pair & (null * 1) ->-> MM' = const(<>).

Так как distr:<<>,X> =0 по определению distr и A f:<> = <> по определению A, следовательно, MM' = R.

Случай 2 (основной): pair & (not * null * 1) ->-> MM' = R.

Ясно, что в этом случае R = R", где DEF R" :: appendl * (mM, MM' * (t1 * 1,2)) по определению формы «условие».

Расписывая mM, получаем формулу для R":

R" = appendl * (A IP * distl * (1 * 1,2),
------f------ -- g --
AA IP * A distl * distr * (t1 * 1,2)).
------- Af ------- ---------h---------

Так как

A* (A IP * distl) = AA IP * A distl

по закону 7 на стр. 318, то R» имеет вид

appendl * (f * g , A f * h)

для указанных под определением R» функций f, g и h.

Поэтому по закону 5

R» = A f * appendl * (g, h) = A f * appendl * ((1*1,2), distr*(t1*l, 2)),

что по закону 6 на стр. 318 дает

A f * distr,

то есть MM'. Теорема доказана.

Каждый из трех использованных в доказательстве общих законов очевиден (следует непосредственно из определения соответствующих функций) и может быть обоснован аналогично закону 6.

3.3. Доказательное программирование методом Хоара

На примере метода Бэкуса видно, как подход к доказательному программированию связан со свойствами ЯП, к которому этот метод применяется (модель Б специально строилась так, чтобы были справедливы достаточно простые законы алгебры программ). Метод Хоара, к изложению которого мы приступаем, также ориентирован на определенный класс ЯП.

Эти ЯП ближе к традиционным (во всяком случае, в них имеются переменные и присваивания). Характерное ограничение состоит в том, что динамическая структура программы в них должна быть хорошо согласована со статической ее структурой. Другими словами, составить представление о процессе выполнения программы должно быть относительно легко по ее тексту.

Иначе о том же можно сказать так: в этих ЯП по структуре знака относительно легко судить о структуре денотата.

Указанное согласование и позволяет реализовать основную идею Тони Хоара – ввести так называемую **дедуктивную семантику языка**, связывающую программные конструкты непосредственно с утверждениями о значениях программных переменных. При этом математической моделью программы служит так называемая **аннотированная программа**, к которой применимы правила преобразования (правила вывода), представляющие, с одной стороны, ее (дедуктивную) семантику, а с другой – математическую модель внешнего мира.

Допустимы, конечно, и иные содержательные истолкования метода Хоара. Например, сами утверждения о свойствах программы, входящие в аннотированную программу, можно считать элементами модели внешнего для программы мира. Существенны не эти различия, а факт, что в методе Хоара, как и в любом методе доказательного программирования, необходимо формально описывать и программу, и требования к ней, и допустимые их связи и преобразования.

Рассмотрим метод Хоара на примере конкретного ЯП и конкретной программы в этом ЯП. Поскольку для нас важны ключевые идеи, а не точное следование классическим текстам, будем вносить в традиционное изложение метода изменения, по нашему мнению, облегчающие понимание сути дела.

3.3.1. Модель Д

В качестве представителя рассматриваемого класса ЯП рассмотрим модель Д очень простого ЯП, который также назовем языком Д. Он очень похож на язык, использованный Дейкстрой в [24].

Начнем с описания синтаксиса языка Д с помощью несколько модифицированной БНФ. И сам синтаксис, и применяемую модификацию БНФ (она соответствует предложениям Вирта и принята в качестве английского национального стандарта) удобно объяснять в процессе пошаговой детализации исходного синтаксического понятия «программа». На каждом шаге детализации при этом указываются все допустимые конкретизации промежуточных синтаксических понятий (абстракций). В зависимости от интерпретации применяемые при этом правила конкретизации можно считать или соотношениями, связывающими классы текстов (так называемые формальные языки), или правила вывода в некоторой порождающей системе (а именно в контекстно свободной грамматике). Ниже выписаны 16 шагов такой детализации (то есть 16 правил модифицированной БНФ).

Синтаксис языка Д

```
программа      = 'begin' { объявление ';' } { оператор ';' } 'end' .
объявление     = ( 'var' | 'arr' ) имя { ',' имя  } .
оператор       = присваивание | развилка | цикл | 'null' .
присваивание   = переменная ':=' выражение .
переменная     = имя [индекс] .
индекс         = '[' выражение ']' .
выражение      = переменная | число | функция .
функция        = имя '(' выражение { ',' выражение } ')' .
развилка       = 'if' { вариант } 'fi' .
цикл           = 'do' { вариант } 'od' .
вариант        = условие '-->' { оператор ';' } .
условие        = выражение (< | <= | = | /= | >= | > ) выражение .
имя            = буква { буква | цифра } .
число          = цифра { цифра } .
буква          = 'a' | 'b' | ... | 'z' .
цифра          = '0' | '1' | ... | '9' .
```

Отличия от оригинальной БНФ сводятся, во-первых, к тому, что выделяются не названия синтаксических понятий (метасимволы), а символы так называемого терминального алфавита (то есть алфавита, из символов которого строятся программы в описываемом языке). В языке Д терминальный алфавит состоит из букв, цифр и символов 'begin', 'end', 'var', 'arr', 'do', 'od', 'if', 'fi', 'null' и некоторых других (скобок, знаков отношений и т. п.). В неясных случаях для выделения тер-

минальных символов применяется апостроф. Во-вторых, применяются круглые, квадратные и фигурные скобки. Круглые – чтобы сгруппировать несколько вариантов конкретизации понятия (несколько альтернатив). Квадратные – чтобы указать на возможность опускать их содержимое. Фигурные – чтобы указать на возможность выписывать их содержимое нуль и более раз подряд в процессе пошаговой детализации (или, как говорят, порождения) конкретной программы.

Некоторый текст признается **допустимой программой** на языке Д тогда и только тогда, когда его можно получить последовательной конкретизацией (то есть породить) по указанным 16 правилам из исходной абстракции «программа». Такой текст называется **выводимым** из метасимвола «программа». Например, текст

```
'begin' 'var' x, i, n;
   x :=M[1]; i := 1;
      'do' i < n --> i := plus(i,1);
         'if M[i] > x --> x := M[i];
         M[i] <= x --> 'null'; 'fi';
      'od';
'end'
```

допустим в языке Д, а если вместо 'plus(i,1) написать «i+1», то получится недопустимый текст (так как «выражение» может быть только «переменной», «числом» или «функцией»). Однако мы позволим себе для наглядности писать «i+1».

Семантика языка Д. Поясним только смысл развилки и цикла. Смысл остальных конструктов традиционен. Для наших целей достаточно интуитивного представления о нем.

Начнем с так называемой операционной семантики развилки и цикла. Другими словами, поясним, как эти конструкты выполняются.

Назовем **состоянием** некоторое отображение переменных программы в их значения. Это отображение частичное, потому что значения некоторых переменных могут быть неопределенными. Вот пример состояния выписанной выше программы (она вычисляет максимум M[i]):

```
<n --> 5, M --> (2,5,6,8,1), x --> 2, i --> 1>
```

Рассмотрим развилку S вида

```
'if'
   P1 --> S1 ,
     ...
   Pn --> Sn
'fi'
```

Полезно учитывать, что смысл конструктов языка Д специально подобран так, чтобы было легче доказывать свойства написанных на нем программ. Как доказывать, рассмотрим в следующем пункте, а пока объясним семантику развилки S.

Пусть S начинает выполняться в некотором состоянии W. Сначала в состоянии W асинхронно (независимо, а возможно и параллельно) вычисляются все Pi. Каждое из них либо нормально завершается и дает истину или ложь, либо завершается отказом (в частности, зацикливанием). Если хотя бы одно Pi дает отказ, то

S завершается отказом. Если все Pi нормально завершаются (состояние W при этом не меняется!), то случайным образом выбирается Si0 – одно из тех и только тех Si, для которых Pi истинно. Результат выполнения этого Si0 в состоянии W – это и есть результат выполнения всей развилки S. Если же все Pi дают ложь, то S завершается отказом.

Рассмотрим цикл S вида

```
'do'
   P1 --> S1 ,
     ...
   Pn --> Sn
'od'
```

Выполнение цикла отличается от развилки тем, что если все Pi дают ложь, то S нормально завершается с состоянием W (то есть его действие в этом случае равносильно пустому оператору). Когда же выбирается Si0, то после его нормального завершения в некотором состоянии Wi0 цикл S снова выполняется в состоянии Wi0. Другими словами, он выполняется до тех пор, пока все Pi не дадут ложь или не возникнет отказ (в последнем случае и весь S завершится отказом).

Вот и все, что нужно знать об (операционной) семантике языка Д, чтобы воспринимать еще один подход к доказательному программированию.

Как видите, семантика языка Д очень «однородная», она симметрична относительно различных вариантов составных конструктов Это помогает рассуждать о программах, написанных на таком языке.

3.3.2. Дедуктивная семантика

Мы уже не раз упоминали о том, что с одними и теми же текстами (сообщениями) можно связывать различный смысл в зависимости от роли, которую они играют при решении конкретных задач. Это справедливо и для текстов программ, написанных на ЯП. С точки зрения исполнителя, будь в этом качестве человек или машина, смыслом программы естественно считать предписываемую этой программой последовательность действий исполнителя. Правила, сопоставляющие программе последовательность действий (в общем случае – класс последовательностей, соответствующих классу аргументов), – это «исполнительская» семантика. Обычно ее называют **операционной** – установившийся, но не слишком удачный термин, «калька» английского operational.

> **Замечание.** Обычно предполагается, что потенциальный исполнитель программы цели своих действий не знает и не должен знать. Если считать, что уровень интеллекта связан со способностью соотносить планируемые (выполняемые) действия с поставленными целями, то потенциальный исполнитель программы начисто лишен интеллекта, абсолютно туп. Однако эта тупость обусловлена такими его потенциальными достоинствами, как универсальность, определенность и быстродействие. Компьютер способен на все, так как «не ведает, что творит», и делает, что «прикажут»; ему «не приходит в голову» действовать по собственной воле, он не тратит ресурсов на оценку разумности программы, а просто выполняет ее! С такой

точки зрения «тупая» операционная семантика вполне оправдана и весьма полезна – она создает прочную основу для взаимодействия авторов, пользователей и реализаторов ЯП.

Было время, когда только операционную семантику и связывали с ЯП. Да и сейчас, когда говорят или пишут о семантике ЯП, чаще всего имеют в виду именно ее (вспомните известные вам описания Фортрана, Бейсика, Си и др.). Знакомя с языком Д, мы также начали с его операционной семантики.

Станем на точку зрения пользователя языка. Нормальный пользователь интеллектом, конечно, обладает, и для него естественно соотносить свои действия со своими целями. Поэтому «исполнительской» семантики ему, как правило, недостаточно. Ему нужна более «интеллектуальная» семантика, лучше помогающая судить о той роли, которую программа в состоянии играть при решении его задач. Операционная семантика обычных ЯП уводит в дебри мелких действий, вместо того чтобы предоставить интегральную (цельную) характеристику связи аргументов и результатов программы.

Форм, в которых можно давать такую характеристику, может быть много. С каждой из них связан свой способ приписывать программе смысл, своя семантика. Вы уже знакомы с таким способом, когда программе сопоставляется отображение ее аргументов в результаты, – это денотационная семантика. Мы займемся еще одной, **дедуктивной семантикой** (иногда ее называют аксиоматической, или логической).

Если операционная семантика предназначена в основном для того, чтобы четко зафиксировать правила поведения исполнителя (тем самым и выразительные возможности пользователя), то дедуктивная семантика предназначена в основном для того, чтобы четко зафиксировать правила поведения пользователя при доказательстве свойств программ. Наиболее интересное из таких свойств – свойство давать определенные результаты при определенных аргументах.

Уточним несколько туманный смысл слова «определенный». Как и прежде, будем называть состоянием программы отображение переменных программы в их значения. Так что состояние меняется при каждом изменении значения какой-либо переменной.

Говорят, что программа P **частично корректна** относительно предусловия Q и постусловия R, если для всякого начального состояния, удовлетворяющего условию Q, заключительное состояние удовлетворяет условию R.

Тот факт, что программа P частично корректна, можно записать с помощью специального обозначения – так называемой **тройки Хоара**

{Q} P {R},

где в скобках явно выписываются соответствующие пред- и постусловия. Корректность называется «частичной» потому, что не гарантируется попадание в заключительное состояние – в общем случае возможен отказ или зацикливание.

Свойство полной корректности **записывается обычно тройкой Хоара с *квадратными скобками***

[Q] P [R],

что означает: «начав с состояния, удовлетворяющего предусловию Q, программа P всегда завершает работу, причем в состоянии, удовлетворяющем постусловию R».

Дедуктивная семантика – **это правила сопоставления каждой программе множества ее троек Хоара**. Следуя Хоару, эти правила представляют обычно в виде логического исчисления (то есть совокупности аксиом и правил вывода), в котором кроме общих аксиом и правил вывода (исчисления предикатов первого порядка) имеются и правила вывода троек Хоара (свои для каждого ЯП). Тем самым каждой программе в ЯП ставится в соответствие ее «дедуктивный» смысл – множество формально выводимых в этом исчислении троек Хоара.

Если такая дедуктивная семантика согласована с операционной семантикой ЯП так, что выводимость тройки гарантирует ее истинность, то в распоряжении пользователя оказывается аппарат формального доказательства свойств программ на рассматриваемом ЯП, например доказательства их частичной или полной корректности.

3.3.3. Компоненты исчисления Хоара

Посмотрим, что же может понадобиться для построения и практического применения дедуктивной семантики ЯП (на примере языка Д и программы вычисления максимального значения элементов одномерного массива, рассмотренной с теми же целями в [1]).

Во-первых, нужно уметь выражать условия на состояния программы. Для этой цели нам послужат обычные логические формулы, в которых в качестве предметных переменных допустимы обозначения объектов программы.

Другими словами, условия на состояния мы намерены выражать на логическом языке первого порядка. При такой договоренности становится возможным выразить следующее свойство программы для вычисления максимума (свойство ее частичной корректности):

```
      { n >= 1 }.
      x:=M[ l]; i:=l;
      'do' i < n --> i:= i + 1;
(Y1)      'if'  M[ i ] > x --> x := M[ i ]  ;
                      M[ i ]  <= x --> 'null'
          'fi';
      'od';
      { x = max (M,n) }
```

Оно говорит о том, что если запустить записанную между пред- и постусловиями программу при n >= 1, то после ее завершения будет истинным условие x = max (M,n), то есть значение переменной x будет равно максимальному элементу массива M при изменении индекса от 1 до n.

При этом отнюдь не утверждается, что программа действительно завершит свою работу (ведь выписано условие частичной, а не полной корректности!).

Кстати, перед нами пример аннотированной программы, то есть программы на «обычном» ЯП, снабженной аннотациями на специальном языке аннотаций.

В нашем случае последний представляет собой логический язык первого порядка с тройками Хоара и другими полезными дополнениями.

Во-вторых, нужно от свойств одних фрагментов программы уметь переходить к свойствам других фрагментов (в частности, соседних или вложенных). Ясно, что это удобно делать далеко не для каждого ЯП. Кажется интуитивно понятным, что следить за свойствами состояния программы относительно легко, если динамика ее исполнения согласована со структурой ее текста. Другими словами, если об исполнении программы удобно рассуждать, «читая» ее текст последовательно, без скачков и разрывов. Этот **принцип согласования статической и динамической структур** программы положен в основу структурного программирования. Язык Д удовлетворяет этому принципу.

В-третьих, чтобы говорить о свойствах фрагментов программы, нужно уметь их обозначать. Поэтому в тройках Хоара следует допустить не только полные программы, но и фрагменты программ. Кроме того, полезно разметить программу, чтобы было удобно ссылаться на отдельные ее точки. Для этого будем пользоваться номерами в круглых скобках. Размеченный фрагмент программы, вычисляющей максимум, примет вид:

```
(1)    x := M[1]; (2) i := 1; (3)
          'do' i < n --> (4) i := i + 1; (5)
             'if M[ i ] > x --> (6) x := M[ i]; (7)
                      M[ i] <= x --> (8) 'null' (9)
             'fi'; (10)
       'od';  (11)
```

Весь этот фрагмент теперь можно обозначить как Ф(1–11) или даже просто (1–11). Условие на состояние программы в точке t обозначим через q(t). Так что предусловие для Ф(1–11) получит обозначение q(l), а постусловие q(11). Для обозначения тождества условий будем применять двойное двоеточие. Например:

```
q(11) :: { x = max(M,n) }.
```

Наконец, *в-четвертых* (самое важное), для каждого языкового конструкта нужно сформулировать **правила вывода** соответствующих троек Хоара. Эти правила естественно называть **дедуктивной семантикой конструкта**, а их совокупность для всего ЯП – дедуктивной семантикой ЯП. Вскоре мы построим такую семантику для языка Д, а пока сделаем несколько предварительных замечаний.

Как уже сказано, для каждого конструкта языка Д нужно сформулировать правило вывода допустимых троек Хоара. Тройки Хоара абсолютны в том смысле, что их истинность не зависит от контекста фрагмента, входящего в тройку (**почему?**). Однако выводить тройки Хоара удобно с помощью условий, характеризующих состояния программы в отдельных точках. Такие «точечные» условия обычно относительны в том смысле, что их истинность (и выводимость) зависит от других точечных условий.

Процесс вывода тройки Хоара, выражающей свойство корректности некоторой программы P, можно представлять себе как вывод последовательных то-

чечных условий, начиная с крайних – пред- и постусловий. При этом тройка считается выведенной, если удастся вывести соответствующее точечное условие на противоположном конце языкового конструкта (в частности, всей программы).

Переход от предшествующего точечного условия к последующему, относящемуся к другому концу некоторого фрагмента программы, полезно представлять себе как «логическое преодоление» этого фрагмента. Поэтому правила вывода точечных условий и троек назовем **правилами преодоления** конструктов.

Содержательно правила преодоления выражают своеобразные законы «символического выполнения» программы. Точнее, это законы последовательного преобразования предикатов (условий), характеризующих состояния программы. Фрагмент программы при таком подходе естественно считать **преобразователем предикатов**.

Это особенно наглядно при преодолении фрагментов в естественном порядке, слева направо. Однако и движение в противоположном направлении может оказаться плодотворным, особенно если и сама программа, и доказательство ее корректности создаются одновременно.

Ведь постусловие выражает цель работы фрагмента программы. Поэтому преодоление завершающего фрагмента программы автоматически формулирует цель для предшествующего фрагмента и т. д., пока не удастся получить условие, выполненное в исходном состоянии. Другими словами, **конструирование программы можно отождествить с поиском преобразователя целевого предиката (постусловия) в исходный предикат (предусловие)**.

В следующем пункте займемся последовательным преодолением уже готового фрагмента программы слева направо (чтобы сосредоточить внимание на сущности и приемах доказательства свойств программ).

3.3.4. Правила преодоления конструктов языка Д

Наша цель – научиться выводить постусловие из предусловия, последовательно преодолевая фрагменты программы. При этом нас будут интересовать не любые фрагменты, а только достаточно крупные языковые конструкты, осмысленные с точки зрения дедуктивной семантики.

> *Замечание.* Это означает, в частности, что верить нашему доказательству нужно как раз «по модулю» доверия к правильности связи между дедуктивной и операционной семантиками преодолеваемых конструктов. Обычно предполагается, что эта связь тщательно проверена. Впрочем, польза от доказательства – не столько в гарантии правильности (в общем случае, конечно, не обеспечиваемой(!); **почему?**), сколько в систематическом исследовании программы с нетривиальной точки зрения. Такое исследование, безусловно, способствует нахождению в программе ошибок.

Преодолеть конструкт – это значит отразить в постусловии влияние выполнения этого конструкта на состояние, то есть на переменные программы. Состояние

после выполнения конструкта в общем случае зависит от состояния перед его выполнением, а также от категории и строения конструкта.

Поэтому вид постусловия нужно связать с видом предусловия, видом преодолеваемого конструкта и, если нужно, с тройками, характеризующими вложенные конструкты.

Дедуктивная семантика присваивания. Начнем, например, преодолевать Ф(1–2) на стр. 327, то есть по q(1) попытаемся построить разумное q(2). Каждый легко напишет

```
q(2) :: ( n >= 1 & x = M[1] ).
```

Но это написано для конкретного предусловия и конкретного оператора присваивания. Как же обобщить и формализовать прием преодоления присваивания, примененный нами только что интуитивно? Ясно, что мы учли результат выполнения конкретного оператора присваивания Ф(1–2) над конкретными объектами программы x и M[1]. Другими словами, учли операционную семантику присваивания.

Результат выполнения состоит в том, что знак x после выполнения оператора присваивания начинает обозначать то же самое, что до его выполнения обозначал знак M[1].

Другими словами, денотат знака M[1] становится денотатом знака x. Итак, если нам до выполнения присваивания что-то известно про денотат знака M[1], то после выполнения присваивания то же самое можно утверждать и про денотат знака x.

Это и есть *основная идея* описания дедуктивной семантики оператора присваивания:

всякое утверждение про значение выражения e в операторе вида

v := e

остается верным после выполнения этого оператора теперь уже по отношению к значению переменной v.

Осталось придумать, как формализовать эту идею в виде правила преобразования логических формул.

Итак, наша задача – перейти от утверждения про e к утверждению про v, причем первое справедливо до присваивания, второе – после. Присваивание меняет значение v (и значение выражения e, если v в него входит). Поэтому в общем случае предусловие само по себе не годится в качестве постусловия. Нужно, во-первых, найти то, что сохраняется при присваивании (найти его **инвариант**), и, во-вторых, отразить действие присваивания на объекты программы.

Ясно, что таким инвариантом служит всякое утверждение B про старое значение выражения e. Если B истинно до присваивания, то останется истинным и после – ведь старое значение выражения e не меняется. Но форма утверждения B должна быть такой, чтобы и после преодоления присваивания B оставалось утверждением именно про *старое* значение выражения e. Ведь если записать его просто в форме B(e), то оно может после присваивания стать ложным – у выражения e может оказаться новое значение (**каким образом?**).

Поэтому обозначим (на метауровне, то есть в наших рассуждениях о свойствах программы) СТАРОЕ значение выражения e незанятой буквой, например Y, и выразим предусловие в форме

(Y = e) => B(Y),

то есть (Y = e) влечет B(Y). При такой форме записи предусловия в нем явно выделена инвариантная часть B(Y). Для аккуратности потребуем, чтобы переменная v не входила в утверждение B. Теперь можно быть уверенными, что B(Y) действительно не зависит от возможного изменения значения переменной v.

Мы теперь готовы выразить, что именно можно утверждать после присваивания. Ведь старый денотат выражения e стал новым денотатом переменной v! Значит, утверждение B останется истинным, если в него вместо знака Y подставить знак v.

Получаем правило вывода

```
(Y = e) = > B(Y)
-------------------- ,
   B( v --> Y )
```

где под горизонтальной чертой изображен результат подстановки знака v в утверждение B вместо всех вхождений знака Y.

Итак, преодоление присваивания состоит из двух шагов, первый из которых содержательный (творческий), а второй – формальный. На первом нужно *найти инвариант* B, характеризующий старое значение выражения e (причем в него не должен входить знак v!). На втором шаге можно формально *применить правило вывода* постусловия.

> **Замечание.** Для предусловий, не содержащих v, тривиальное правило преодоления присваивания состоит в простом переписывании предусловия в качестве постусловия. Подобные правила полезно применять при доказательстве таких свойств программ, на которые преодолеваемые операторы не могут повлиять в принципе. Однако интереснее, конечно, правило преодоления, существенно учитывающее операционную семантику преодолеваемого оператора. Такими правилами мы и занимаемся.

Применение правила преодоления присваивания. Попытаемся двинуться по нашей программе-примеру, стараясь преодолеть оператор (1–2) на стр. 327 и получить «интуитивно» написанное нами постусловие q(2) теперь уже формально.

Во-первых, нужно подобрать подходящее B. Как уже было объяснено, это задача творческая. Зная q(2), можно догадаться, что B должно иметь вид

{n>=1 & Y=M[1] }.

> **Замечание.** Вот так знание желательного постусловия (по существу, знание цели выполняемых действий) помогает формально преодолевать конструкты программы.

Нам нужно вывести из q(1) обычными логическими средствами предусловие для преодоления оператора (1–2). Другими словами, подготовить предусловие для формального преодоления оператора.

Предусловие должно иметь вид

(Y = M[1]) ==> (n >= 1 & Y= M[1]).

Оно очевидно следует из n >= 1. Нужно формально применить схему аксиом

(A => (C => A & C)

и правило вывода модус поненс

```
A, A => B
------------ .
     B
```

Подставляя вместо A утверждение (n>=1), а вместо C – утверждение (Y=M[1]), получаем нужное предусловие. Итак, все готово для формального преодоления фрагмента (1–2) с помощью правила преодоления присваивания.

Совершенно аналогично нетрудно преодолеть и фрагмент (2–3) на стр. 327, получив

q(3) :: (n >= 1 & x = M[1] & i = 1).

> **Замечание.** Нетривиальность первого из этапов преодоления оператора присваивания подчеркивает принципиальное отличие дедуктивной семантики от операционной. Дедуктивная семантика не предписывает, а разрешает. Она выделяет законные способы преодоления конструктов, но не фиксирует жестко связь предусловия с постусловием. Зато она позволяет преодолевать один и тот же оператор по-разному, выводя разные постусловия в зависимости от потребностей того, кто выясняет (или доказывает) свойства программы. Можете ли вы привести пример постусловия для (2–3), отличного от q(3)?

Перепишем наше правило преодоления присваивания, обозначив через L предусловие, а через R – постусловие:

```
              L :: (Y = e ) => B(Y)
П(1)          ------------------------ .
                R :: B( v --> Y)
```

Чтобы преодолеть конструкт (3–8) на стр. 327, нужно запастись терпением и предварительно разобраться с дедуктивной семантикой остальных конструктов языка Д.

Дедуктивная семантика развилки. Рассмотрим развилку S вида

```
'if'
   P1 --> S1
     ...,
   Pn --> Sn
'fi'
```

Наша задача – формализовать для нее правила преодоления. Вспомним, что по смыслу (операционной семантике) развилки каждая ее i-я ветвь Si выполняется только тогда, когда истинен соответствующий предохранитель Pi, причем завершение Si означает завершение всей развилки S. Так как по определению постусловия оно должно быть истинным после выполнения любой ветви, получаем следу-

ющее естественное правило преодоления, сводящее преодоление S к преодолению ее ветвей:

$$(П2)\qquad \frac{\forall k : \{L \,\&\, Pk\}\ Sk\ \{R\}}{R},$$

где L и R – соответственно пред- и постусловия для S.

Таким образом, преодоление развилки следует осуществлять разбором случаев, подбирая такое R, чтобы оно было истинным в каждом из них. Очень часто R представляет собой просто дизъюнкцию постусловий Rl v...v Rn для операторов Sl,...,Sn соответственно. Подчеркнем, что преодоление развилки невозможно, если не выполнено ни одно условие Ri.

Дедуктивная семантика точки. Поскольку наша цель – научиться формально преобразовывать утверждения о программе в соответствии с ее операционной семантикой, то естественно считать допустимой замену утверждения, привязанного к некоторой точке программы, любым его чисто логическим следствием, привязанным к той же точке. Для единообразия можно считать точку пустым фрагментом (фрагментом нулевой длины), а произвольное чисто логическое правило вывода – правилом преодоления пустого фрагмента. Применение таких правил очень важно – с их помощью готовят преодоление непустых конструктов программы (мы уже действовали таким способом при преодолении фрагмента (1–2)). Таким образом, дедуктивная семантика точки совпадает с дедуктивной семантикой пустого фрагмента. Такова же и дедуктивная семантика оператора «null».

Дедуктивная семантика цикла. Рассмотрим цикл вида

```
'do'
   P1 -->S1,
     ...
   Pn --> Sn
'od'
```

Наша задача – сформулировать правило его преодоления. Вспомним операционную семантику этого оператора. Он завершает исполнение тогда и только тогда, когда истинно ¬P1 & ... & ¬Pn. Обозначим эту конъюнкцию отрицаний через P и немного порассуждаем о природе циклов.

Циклы – важнейшее средство для описания *потенциально неограниченной совокупности действий ограниченными по длине предписаниями*. Таким средством удается пользоваться в содержательных задачах только за счет того, что у всех повторений цикла обнаруживается некоторое общее свойство, **инвариант цикла**, *не меняющийся от повторения к повторению*.

В языке Д выполнение циклов состоит только из повторений тела цикла, поэтому инвариант цикла должен характеризовать состояние программы как непосредственно перед началом работы цикла, так и сразу по его завершении.

Обозначим инвариант цикла через I. Естественно, у одного цикла много различных инвариантов (**почему?**). Тем не менее основную идею цикла, отражающую его роль в конкретной программе, обычно удается выразить достаточно полным инвариантом I и условием завершения P.

Условие P отражает достижение цели цикла, а конъюнкция I & P – свойство состояния программы, достигнутого к моменту завершения цикла. Значит, это и есть постусловие для цикла S. А предусловием служит, конечно, инвариант I – ведь он потому так и называется, что истинен как непосредственно перед циклом, так и непосредственно после каждого исполнения тела цикла. Осталось выразить сказанное формальным правилом преодоления:

(П3) $$\frac{I}{I \;\&\; P}.$$

Это изящное правило обладает тем недостатком, что в нем формально не отражена способность утверждения I служить инвариантом цикла. Нужно еще явно потребовать его истинности после каждого исполнения тела (или, что то же самое, после исполнения каждой ветви). Получаем следующее развернутое правило преодоления:

(П4) $$\frac{\forall k : \{I \;\&\; Pk\}\ Sk\ \{I\}}{I \;\&\; P}.$$

Другими словами, если утверждение I служит инвариантом цикла, то есть для каждого Pk истинность I сохраняется при выполнении k-й ветви цикла, то результатом преодоления всего цикла может служить постусловие I & P.

Скоро мы продолжим движение по нашей программе с использованием инвариантов цикла. Но прежде завершим построение дедуктивной семантики языка Д.

От точечных условий к тройкам. Нетрудно заметить, что как в правиле (П2), так и в (П4) предусловиями служат не точечные условия, а тройки Хоара. Поэтому требуется формальное правило перехода от точечных условий к тройкам. Оно довольно очевидно. В сущности, именно его мы имели в виду, объясняя саму идею преодоления фрагментов.

Зафиксируем некоторый фрагмент Ф и обозначим через L(Ф) некоторое точечное условие для его левого конца, а через R(Ф) – некоторое точечное условие для его правого конца. Через «!==>» обозначим отношение выводимости с помощью наших правил преодоления. Получим

(П5) $$\frac{L(\Phi)\ !{=}{=}{>}\ R(\Phi)}{\{L\}\ \Phi\ \{R\}}.$$

Замечание. Может показаться, что это правило не совсем естественное, и следовало бы ограничиться только правильными языковыми конструктами, а не заниматься любыми фрагментами. Действительно, достаточно применять это правило только для присваиваний, ветвлений, циклов и последовательностей операторов. Но верно оно и в том общем виде, в котором приведено (**почему?**). При этом недостаточно, чтобы точка привязки утверждения L текстуально предшествовала точке привязки R. Нужна именно выводимость в нашем исчислении (**почему?**).

Итак, мы завершили построение исчисления, фиксирующего дедуктивную семантику языка Д.

3.3.5. Применение дедуктивной семантики

Теперь мы полностью готовы к дальнейшему движению по нашей программе-примеру. Предстоит преодолеть цикл (3–11) на стр. 327, исходя из предусловия q(3) и имея целью утверждение q(11). Подчеркнем в очередной раз, как важно понимать цель преодоления конструктов (легко, например, преодолеть наш цикл, получив постусловие n>=1, но нам-то хотелось бы q(11)!).

Правило преодоления цикла требует инварианта. Но нам годится не любой инвариант, а только такой, который позволил бы в конечном итоге вывести q(11). Интуитивно ясно, что он должен быть в некотором смысле *оптимальным* – с одной стороны, выводимым из q(3), а с другой – позволяющим вывести q(11). Обычная эвристика при поисках такого инварианта – *постараться полностью выразить в нем основную содержательную идею рассматриваемого цикла.*

Замечание. Важно понимать, что разумные циклы преобразуют хотя бы некоторые объекты программы. Поэтому инвариант должен зависеть от переменных (принимающих, естественно, разные значения в процессе выполнения цикла). Однако должно оставаться неизменным фиксируемое инвариатом соотношение между этими значениями.

Внимательно изучая цикл (3–11) на стр. 327, можно уловить его идею – при каждом повторении поддерживать x равным max(M,i), чтобы при i = n получить q(11). Выразим этот замысел формально

I1 :: (x = max(M,i)

и попытаемся с помощью такого I1 преодолеть наш цикл.

Замечание. «Вылавливать» идеи циклов из написанных программ – довольно неблагодарная работа. Правильнее было бы формулировать инварианты при проектировании программы, а при доказательстве пользоваться заранее заготовленными инвариантами. Мы лишены возможности так действовать, потому что само понятие инварианта цикла появилось в наших рассуждениях лишь недавно. Однако и у нашего пути есть некоторые преимущества. По крайней мере, есть надежда почувствовать сущность оптимального инварианта.

Предстоит решить три задачи:

1. Доказать, что I1 – действительно инвариант цикла (3–11).
2. Доказать, что условие q(11) выводимо с помощью I1.
3. Доказать, что из q(3) логически следует I1.

Естественно сначала заняться двумя последними задачами, так как наша цель – подобрать оптимальный инвариант. Если с помощью I1 нельзя, например, вывести q(11), то им вообще незачем заниматься. Так как задача (в) тривиальна при i = 1, займемся задачей (б).

Замечание. На самом деле задача (в) тривиальна лишь при условии, что можно пользоваться формальным определением функции max (точнее, определяющей эту функцию системой соотношений-аксиом). Например, такими соотношениями:

∃k : (k>=1) & (k<=i) & M[k]=max(M,i);

$\forall k : (k>=1) \ \& \ (k<=i) => M[k] <= max(M,i)$.

При i = 1 отсюда следует M[1]=max(M,1). Так что I1 превращается в (x= [1]), то есть просто в одну из конъюнкций q(3).

По сути, это замечание привлекает внимание к факту, что при доказательстве правильности программ методом Хоара приходится все используемые в утверждения понятия описывать на логическом языке первого порядка и непосредственно применять эти (довольно громоздкие) описания в процессе преодоления конструктов. Сравните с методом Бэкуса.

Первая попытка решить задачу (б). Итак, допустим, что I1 – инвариант цикла, и попробуем вывести q(11). По правилу преодоления (П4) в точке (I1) выводимо

$q(11)a :: x = max(M,i) \ \& \ \neg (i < n)$.

Сразу ясно, что q(11) не выводимо из q(11)a. Легко построить противоречащий пример:

$i = 3, n= 2, M = (1,3,10); max (M,3) = 10$.

Корректировка инварианта. Как видно, мы не зря сразу занялись задачей (б). Придется внимательнее изучить цикл и понять, что мы упустили, формируя его инвариант.

Контрпример получен при i > n. Ясно, что в цикле (3–11) такое значение i получиться не может, он сохраняет условие i <= n. Но ведь это значит, что обнаружен еще один претендент на роль инварианта цикла! Обозначим его через I2

$I2 :: i <= n$.

Нетрудно проверить, что, соединяя I1 с I2 в утверждении

$I3 :: I1 \ \& I2$,

можно доказать q(11). Проведем это доказательство.

Действительно, если I3 окажется инвариантом, то по правилу преодоления цикла выводимо для точки (11)

$q(11)b :: I1 \ \& \ I2 \ \& \ \neg (i < n)$.

Но

$q(11)b => (x = max(M,i) \ \& \ (i = n) => x= max(M,n)$.

Что и требовалось.

Правило соединения инвариантов цикла. Уместно отметить, что «пополнять» утверждения, претендующие на роль инварианта, приходится довольно часто в процессе подбора оптимальных инвариантов. Поэтому полезно сформулировать общее правило:

Конъюнкция инвариантов некоторого цикла остается инвариантом этого цикла.

Обратное, естественно, неверно. **(Приведите контрпример.)**

Это правило более общего характера, чем правила преодоления языка Д, оно справедливо для любых инвариантов любого преодоления конструктов любого ЯП.

Инвариантность I1 и I2. Опираясь на правило соединения инвариантов, мы можем теперь решать задачу (а) отдельно для I1 и I2. Займемся сначала доказательством инвариантности I2 как делом более простым.

Напомним, что доказать инвариантность I2 для цикла (3–11) – это значит доказать истинность утверждения

∀k : {I2 & Pk} Sk {I2},

которое в нашем случае сводится к единственной тройке Хоара

{I2 & (i < n)} Ф(4–10) {I2},

так как в цикле (4–11) со стр. 327 лишь один вариант. Чтобы вывести нужную тройку, начнем с утверждения

q(4)a :: I2 & (i < n) :: (i <= n) & (i < n)

как предусловия для Ф(4–11) и постараемся применить правила преодоления сначала присваивания (4–5) со стр. 327, а затем развилки (5–10) со стр. 327 для вывода утверждения q(10)a :: I2.

Но

q(4)a => (i < n),

и по правилу преодоления присваивания получаем

!=> (i <= n) :: q(5)a.

Аккуратный вывод q(5)a предоставляем читателю (достаточно подготовить для Ф(4–5) предусловие в виде (Y= i+1) => (Y <= n)).

Теперь одного взгляда на фрагмент (5–10) достаточно, чтобы убедиться, что он сохраняет q(5)a – ведь он не изменяет ни i, ни n. Но это соображения содержательные, а при формальном выводе несложно воспользоваться правилами преодоления развилки и вложенных в него операторов (присваивания и пустого). Оставим это в качестве упражнения и закончим тем самым доказательство инвариантности I2.

Внешний инвариант. Полезно сформулировать в явном виде правила преодоления для утверждений, не зависящих от объектов, изменяемых в преодолеваемых фрагментах. При этом мы, конечно, не получим принципиально новых возможностей преодоления. Однако упрощенные правила бывают особенно удобны при преодолении «по частям», которым мы только что воспользовались (разбив инвариант цикла на части и занимаясь ими по очереди).

Ясно, что упрощенное правило преодоления должно состоять в переписывании предусловия в качестве (конъюнктивного члена) постусловия.

Назовем **внешним инвариантом** преодолеваемого фрагмента всякое утверждение, к которому применимо такое упрощенное правило. Сформулировать признаки внешних инвариантов для отдельных конструктов языка Д – полезное упражнение.

Инвариантность I1. Вернемся к нашей программе-примеру на стр. 327 и попытаемся доказать, что I1 – инвариант цикла (3–11).

Нужно доказать утверждение

∀k : {I1 & Pk} Sk {I1},

то есть в нашем случае

{x = max(M,i) & (i < n)} Ф(4-10) {x = max(M,i)}.

Обозначим I1 & (i < n) через q(4) и рассмотрим его как предусловие для присваивания (4–5).

Преодолев присваивание, получим

q(5) :: (x = max(M.i-1)) & (i-1 < n).

Чтобы выполнить это преодоление аккуратно по правилам, нужно сначала применить правило преодоления точки и получить

(Y = i+1) => (x = max(M,Y-1)) & (Y-l,n),

то есть получить предусловие присваивания в удобной для преодоления форме, а затем получить q(5) непосредственно по правилу (П1).

Теперь нужно преодолеть развилку (5–10) на стр. 327.

В соответствии с правилом (П2) постусловие развилки должно быть постусловием каждой ветви развилки. Нам нужно получить в качестве такового I1. Со второй ветвью развилки (5–10) никаких сложностей не возникает:

q(5) & (M[i] <= x) => (x = max(M,i)) :: I1.

(Применено правило преодоления пустого оператора, то есть обычное логическое следование.)

Займемся первой ветвью. Ясно, что предусловие

q(6) :: q(5) & (M[i] > x)

непосредственно непригодно для преодоления присваивания (6–7) на стр. 327. Формально потому, что зависит от x. Содержательно потому, что связь «нового» рассматриваемого значения массива M[i] с остальными значениями (проявляющаяся в том, что M[i] – максимальное из них) выражена неявно и к тому же через значение x, которое «пропадает» в результате преодоления присваивания. Так что наша ближайшая цель – в процессе подготовки к преодолению проявить эту связь. Именно

q(6) => (M[i] = max(M.i) :: q(6)b.

Обозначив M[i] через Y, нетрудно теперь вывести I1 в качестве постусловия первой ветви развилки (5–10), а следовательно, и цикла (3–11).

Осталось убедиться, что 13 логически следует из q(3). Это очевидно.

Исключение переменных. Подчеркнем важность приема, примененного при преодолении присваивания (6–7) на стр. 327, точнее методологическое значение этого приема при доказательстве свойств программ на основе дедуктивной семантики. *Перед преодолением операторов, содержательно влияющих на предусловие, необходимо вывести из него логическое следствие, не зависящее от изменяемых переменных* (то есть найти инвариант). Назовем этот прием **исключением переменных**.

Получение подходящих следствий предусловия – творческий акт в преодолении таких операторов. Методологическое значение приема исключения переменных сопоставимо со значением творческого подбора инвариантов цикла. Так что методика применения дедуктивной семантики для доказательства корректности программ довольно тонко сочетает творческие содержательные действия с чисто формальными.

Подведем итоги раздела. Мы провели доказательство содержательного утверждения о конкретной программе на языке Д, пользуясь его дедуктивной семантикой и рядом методических приемов, опирающихся на понимание сути этой программы. Однако проверить корректность самого доказательства можно теперь чисто формально, не привлекая никаких содержательных (а значит, подозрительных по достоверности) соображений. Достаточно лишь в каждом случае указывать соответствующее формальное правило преодоления конструкта и проверять корректность его применения. Итак, мы построили дедуктивную семантику языка Д и разработали элементы методики ее применения.

Вопрос. Могут ли в программе вычисления максимума остаться ошибки? Если да, то какого характера?

Вопрос. Видите ли вы в доказательном программировании элементы, характерные для взгляда на ЯП с математической позиции? Какие именно? Чем с этой позиции отличаются методы Бэкуса и Хоара?

Глава 4

Реляционное программирование (модель Р)

4.1. Предпосылки 340
4.2. Ключевая идея 341
4.3. Пример 341
4.4. О предопределенных отношениях 347
4.5. Связь с моделями МТ и Б 348

4.1. Предпосылки

Основную идею классического операционного (процедурного) подхода к программированию можно сформулировать следующим образом.

Для каждого заслуживающего внимания класса задач следует придумать алгоритм их решения, способный учитывать параметры конкретной задачи.

Записав этот алгоритм в виде программы для подходящего исполнителя, получим возможность решать любую задачу из рассматриваемого класса, запуская созданную программу с подходящими аргументами.

Итак, исходными понятиями операционного подхода служат:

- класс задач;
- универсальный алгоритм решения задач этого класса;
- параметрическая процедура, представляющая этот алгоритм на выбранном исполнителе;
- вызов (конкретизация) этой процедуры с аргументами, характеризующими конкретную задачу. Исполнение этого вызова и доставляет решение нужной задачи.

Конечно, это весьма упрощенная модель «операционного мышления». Достаточно вспомнить, что, например, понятие параллелизма заставляет отказаться от представления о единой процедуре, определяющей последовательность действий исполнителя, и ввести понятие асинхронно работающих взаимодействующих процессов. Однако сейчас для нас главное в том, что каждый процесс остается по существу параметрической процедурой, способной корректировать последовательность действий исполнителя в зависимости от характеристик решаемой задачи. И не важно, передаются ли эти характеристики в качестве аргументов при запуске процесса, извлекаются им самостоятельно из программной среды или определяются при взаимодействии с другими процессами.

Важно понимать, что в операционном подходе центральным понятием, характеризующим класс задач, считается алгоритм (процедура) их решения. Другими словами (в другой терминологии), можно сказать, что параметрический алгоритм представляет знания об этом классе задач в процедурной (иногда говорят «рецептурной») форме. Такие знания, в свою очередь, служат абстракцией от конкретной задачи. Аргументы вызова алгоритма представляют знания уже о конкретной решаемой задаче.

Так что операционный подход требует представлять знания о классе задач сразу в виде алгоритма их решения, позволяя абстрагироваться лишь от свойств конкретной задачи. Между тем жизненный опыт подсказывает, что любой осмысленный класс задач характеризуется прежде всего определенными знаниями о фактах, понятиях и соотношениях в той проблемной области, к которой относится этот класс.

Например, прежде чем сочинять процедуру, способную вычислять список всех племянников заданного человека, нужно знать, что такое «сестра», «брат», «родитель» и т. п. Причем эти знания вовсе не обязаны быть процедурными – они могут

касаться вовсе не того, КАК вычислять, а, например, того, ЧТО именно известно о потенциальных исходных данных и результатах таких вычислений.

Такие знания часто называют «непроцедурными», желая подчеркнуть их отличие от процедурных. Однако если стараться исходить из собственных свойств такого рода знаний, то, заметив, что они касаются обычно фактов и отношений между объектами проблемной области, лучше называть их «логическими», или «реляционными» (от англ. *relation* – отношение).

4.2. Ключевая идея

Важно заметить, что если бы удалось отделить реляционные знания от процедурных, возникла бы принципиальная возможность освоить новый уровень абстракции со всеми вытекающими из этого преимуществами для технологии решения задач. Ведь реляционное представление знаний о классе задач – абстракция от способа (алгоритма, процедуры) решения этих задач (и, следовательно, может обслуживать самые разнообразные такие способы).

Более того, возникает соблазн разработать универсальный способ решения произвольных задач из некоторой предметной области, параметром которого служит реляционное представление знаний об этой области.

Это и есть ключевая идея реляционного подхода. Другими словами, в этом подходе привычное программирование как деятельность по созданию алгоритмов и представлению знаний о них в виде программ процедурного характера становится совершенно излишним. Программирование сводится к представлению реляционных знаний о некоторой предметной области (например, о родственных отношениях людей). Такое представление совместно с представлением данных о конкретной задаче из рассматриваемой области (например, указанием конкретного человека) служит аргументом для универсального алгоритма, выдающего решение этой конкретной задачи (например, список племянников указанного человека).

Основное достижение – в том, что переход к новой задаче (который при традиционном подходе потребовал бы создания новой программы), например к задаче о списке всех родных теток заданного человека, не потребует никакого программирования! Достаточно правильно сформулировать задачу (то есть правильно представить в реляционном стиле знания о ее исходных данных и ожидаемых результатах).

Конечно, чтобы она оказалась практичной, нужно выполнить целый ряд требований. К ним мы еще вернемся.

4.3. Пример

Представим реляционные знания о родственных отношениях. Другими словами, опишем «мир» родственных отношений. Содержательно это будет, конечно, очень упрощенная модель реального мира человеческих отношений.

4.3.1. База данных

```
ф1)  (мужчина, Иван) — Иван — мужчина
ф2)  (мужчина, Степан)
ф3)  (мужчина, Николай)
ф4)  (мужчина, Кузьма)
ф5)  (женщина, Марья) — Дарья — женщина
ф6)  (женщина, Дарья)
ф7)  (родитель, Степан, Николай) — Степан — родитель Николая
ф8)  (родитель, Дарья, Кузьма) — Дарья — родитель Кузьмы
ф9)  (родитель, Иван, Дарья)
ф10) (родитель, Иван, Степан)
```

Итак, мы пользуемся простейшим языком представления реляцонных знаний. Представлены три конечных отношения – «мужчина», «женщина» и «родитель». Два первых – одноместные (унарные), третье – двухместное (бинарное). Отношение представлено конечным множеством кортежей, каждый из которых представляет элемент отношения – элементарный факт, касающийся некоторых имен-атомов. При этом имя отношения всегда занимает первую позицию в кортеже. Позиция атома в кортеже, конечно, существенна.

Например, (родитель, Дарья, Кузьма) и (родитель, Кузьма, Дарья) представляют разные факты.

Совокупность отношений называется реляционной **базой данных** (БД).

4.3.2. База знаний

Вместе с тем содержательный смысл отношений пока никак нами не представлен. Его можно проявить только за счет указания связей между отношениями! Представим некоторые из таких связей так называемыми **предложениями**. Содержательно предложения служат правилами вывода, позволяющими строить одни отношения из других.

Определим правила вывода отношений «брат», «сестра», «общий_родитель», «дядя» и «тетя».

```
п1) (брат, X, Y) (мужчина, X) (общие_родители, X, Y)
п2) (сестра, X, Y) (женщина, X) (общие_родители, X, Y)
п3) (общий_родитель,X,У) (родитель,Z,X) (родитель,Z,Y)
п4) (дядя, X, Y) (мужчина, X) (родитель, Z,Y) (брат, X, Z)
п5) (тетя, X, Y) (женщина, X) (родитель, Z,Y) (сестра, X, Z)
```

Формально предложение – это кортеж кортежей, в которых допускаются не только атомы, но и переменные. Переменные будем обозначать большими латинскими буквами. Совокупность предложений называется **базой знаний** (БЗ) (иногда этим термином называется совокупность предложений вместе с БД; во всяком случае, именно наличие правил вывода отличает базу знаний от базы данных).

Нетрудно догадаться, что предложения позволяют выводить новые факты из фактов, уже содержащихся в БД. Например, можно вывести факты

```
(общие_родители, Степан, Дарья)
   (общие_родители, Дарья, Степан)
      (брат, Степан, Дарья)
```

4.3.3. Пополнение базы данных (вывод фактов)

Точный смысл правил вывода (семантику реляционного языка) можно объяснять по-разному. Начнем с метода, никак не учитывающего конкретную задачу, которую предполагается решать. Назовем его **разверткой** БД. Сформулируем сначала суть развертки, а потом продемонстрируем ее на примере нашей БЗ.

Суть развертки. Первый кортеж каждого правила интерпретируется как «следствие» из «условий», представленных остальными кортежами этого правила. Наглядно это можно выразить формулой

T <== B1&...&Bn,

где T – первый кортеж (следствие, теорема), а Bi – условия (посылки, аксиомы, факты).

Цель развертки: построить БД, содержащую все факты, выводимые из фактов исходного состояния БД посредством правил вывода из БЗ.

Полная развертка состоит из последовательности циклов, в каждом из которых каждое предложение поочередно применяется к текущему состоянию БД. Вначале текущим состоянием считается исходное состояние БД.

Очередное применение предложения состоит из последовательности всех разверток, выполняемых этим предложением при определенной подстановке атомов вместо переменных (поскольку число переменных и атомов конечно, то и число таких разверток конечно; вместо каждой переменной подставляется один и тот же атом).

Развертка состоит в том, что если все кортежи-условия содержатся в соответствующих отношениях текущего состояния БД, то кортеж-следствие (после замены в нем переменных атомами) пополняет соответствующее отношение (если его еще там нет).

Развертка завершается, когда очередной цикл не добавляет ни одного нового кортежа ни в одно отношение.

Заметим, что развертка завершается при любой исходной БД. **(Почему?)**

Рассмотрим пример.

Первая развертка правил **(п1)** и **(п2)** со стр. 342 пуста (так как отношение «общие родители» пусто). Первая развертка правила **(п3)** при Z=Иван пополняет отношение «общие_родители» кортежами

```
(общие_родители, Степан, Дарья) и
(общие_родители, Дарья, Степан).
```

Первая развертка правил **(п4)** и **(п5)** со стр. 342 также пуста (так как отношения «брат» и «сестра» пока по-прежнему пусты).

Во втором цикле «общие_родители» уже не пусто, и развертка правила **(п1)** добавляет кортеж

```
(брат, Степан, Дарья),
```

а правило **(п2)** добавляет кортеж

```
(сестра, Дарья, Степан).
```

В этом же цикле развертка правил **(п4)** и **(п5)** добавляет кортежи

```
(дядя, Степан, Кузьма)
(тетя, Дарья, Николай).
```

Так как третий цикл ничего нового не добавляет, развертка завершается.

Теперь все готово для решения конкретных задач из предметной области, знание о которой представлено БЗ.

4.3.4. Решение задач

Конкретная задача формулируется в виде кортежа (обычно с переменными), выделяемого знаком вопроса. Например:

```
?(дядя, Q, Кузьма).
```

Вопрос рассматривается в качестве образца, для которого требуется подобрать кортежи из отношений БД, получаемые из образца подходящей заменой переменных. Решением задачи считается перечень всех таких кортежей. Например, решение нашей задачи имеет вид

```
(дядя, Степан, Кузьма).
```

Содержательный смысл решения очевиден (спрашивается, кто дядя Кузьмы; ответ: Степан). Понятно, что несложно выдавать ответ и в виде, например,

```
Q = Степан.
```

Нетрудно понять, что таким образом можно решить любую задачу из «мира родственных отношений» Ивана, Степана, Николая, Кузьмы, Марьи и Дарьи.

Например:

```
?(тетя, R, Николай)    R = Дарья
?(Q, Степан, Дарья)    Q = общие_родители или Q = брат.
```

Итак, показано, как можно представить реляционные знания для целого класса задач таким образом, что решение конкретной задачи не требует никакого программирования. Человек описывает мир на языке представления знаний, затем человек ставит задачу на языке запросов, а компьютер дает решение задачи, пользуясь универсальным решающим алгоритмом (в нашем случае это алгоритм развертки). Отличие от обычной реляционной БД – в БЗ, написанной на языке представления знаний.

4.3.5. Управление посредством целей

Если до сих пор мы стремились лишь объяснить семантику реляционного языка, то теперь пришло время подумать о его эффективности. Бросается в глаза, что

развертка слишком расточительна с точки зрения потребностей конкретных задач. Для ответа на запрос о дяде Кузьмы совершенно не требуются отношения «сестра» и «тетя», которые тем не менее и вычисляются, и хранятся в БД. Другими словами, развертка готовит ответы сразу на все случаи жизни, чего нельзя себе позволить в реальных условиях.

Суть управления посредством целей. Поищем иной принцип использования исходной БЗ, с тем чтобы по возможности делать лишь ту работу, которая необходима для решения конкретных задач.

Ясно, что лишняя работа делается из-за того, что развертка никак не использует постановку задачи (и даже ничего не «знает» о ней). Ключевая идея нового принципа использования БЗ в том и состоит, чтобы при попытке ответить на запрос анализировать те и только те правила из БЗ, которые могут оказаться полезными именно для этого запроса, этот «новый» принцип нам в сущности уже хорошо знаком – это принцип пошаговой детализации «сверху вниз» – от исходной задачи к подзадачам (от исходной цели к подцелям).

Главное при управлении посредством целей – уметь выбирать такие подцели, которые действительно способствуют достижению цели верхнего уровня, и вовремя прекращать заниматься подцелями, которые оказались бесперспективными (*с точки зрения цели верхнего уровня*).

Уточнения и примеры. Постановка задачи считается первой текущей целью-запросом. Затем БД и БЗ совместно используются для ответа на запрос. При этом последовательно анализируются следствия (первые кортежи) предложений БЗ и факты БД – тривиальные следствия. Следствие считается сопоставимым с запросом-целью, если существует такая **согласующая подстановка** (значений вместо переменных запроса и следствия), в результате которой запрос совпадает со следствием.

Например, следствие (дядя, X, Y) сопоставимо с запросом (дядя, Q, Кузьма), так как они совпадают после согласующей подстановки

```
X -> Q, Y -> Кузьма.
```

Дерево целей. Если найденное сопоставимое следствие оказывается фактом (то есть не содержит переменных), то цель считается достигнутой. Если же сопоставимое следствие начинает некоторое предложение, то условия из этого предложения становятся подцелями. Говоря точнее, подцелями становятся не сами условия, а результат применения к ним согласующей подстановки.

Например, из цели (дядя, Q, Кузьма) образуются связанные подцели

```
(мужчина, Q) (родитель, Z, Кузьма)
(брат, Q, Z).
```

Обратите внимание на замену переменных в подцелях по сравнению с исходными условиями. Связанность этих подцелей проявляется в том, что цель верхнего уровня может считаться достигнутой только при условии, что ее непосредственные подцели достигаются *совместно*, при одной и той же согласующей подстановке.

Например, для цели (мужчина, Q) сопоставимым следствием оказывается факт (мужчина, Иван) при согласующей подстановке

```
Q -> Иван.
```

А для цели (родитель, Z, Кузьма) сопоставимым следствием – факт (родитель, Дарья, Кузьма) при согласующей подстановке

```
Z -> Дарья.
```

Тогда для достижения исходной цели и третья подцель должна быть достижима при подстановке

```
Q -> Иван, Z -> Дарья.
```

Однако нетрудно убедиться, что подцель (брат, Иван, Дарья) не может быть достигнута.

Действительно, из **(п1)** со стр. 342 возникает новая совокупность подцелей

```
(мужчина, Иван) (общие_родители, Иван, Дарья),
```

а затем из **(п3)** –

```
(родитель, Z, Иван) (родитель, Z, Дарья).
```

Однако ни для какого Z в БД нет факта, сопоставимого с первой из этих подцелей.

Итак, принципиально важный момент – что делать, когда для некоторой подцели найти согласующую подстановку не удается. Назовем такую ситуацию **тупиком**.

Тупики и перебор с возвратом. Конечно, такую подцель следует признать недостижимой, так как для нее проанализированы все потенциально сопоставимые следствия. Однако не исключено, что цель верхнего уровня все-таки достижима. Ведь недостижимость конкретной ее подцели могла быть вызвана неудачным выбором либо подстановок в связанных подцелях, либо подстановки при переходе от цели верхнего уровня к подцелям.

Например, для цели (мужчина, Q) другие сопоставимые следствия-факты – (мужчина, Степан), (мужчина, Николай) и (мужчина, Кузьма). Причем каждому из них соответствует своя согласующая подстановка.

Проблема тупиков в дереве подцелей решается классическим методом – так называемым **перебором с возвратом** (backtracking) потенциальных сопоставимых следствий и согласующих подстановок. Для его реализации нужно организовать так называемый **стек возвратов**, где и запоминать место сопоставимого следствия вместе с соответствующей согласующей подстановкой, с тем чтобы иметь возможность продолжить поиск согласующей подстановки, когда возникнет тупик.

Например, в нашем случае придется вернуться к подцели (мужчина, Q) и выбрать другое следствие-факт (мужчина, Степан) при новой согласующей подстановке

```
Q -> Степан.
```

Тогда при той же подстановке с

```
Z -> Дарья
```

в качестве третьей подцели получим

```
(брат, Степан, Дарья),
```

что выводимо посредством **(п1)** с учетом **(ф2)**, **(п3)**, **(ф9)** и **(ф10)**.

Итак, исходная цель будет достигнута при Q = Степан и тем самым получено решение задачи (обратите внимание: правило **(п5)** не было использовано).

> **Замечание.** Управление посредством целей описано нами в значительной степени в традиционном операционном стиле, хотя, конечно, был соблазн применить реляционный стиль. Однако это именно соблазн, потому что даже если игнорировать проблемы читателя, которому о новом для него принципе рассказывают, опираясь на сам этот новый принцип, останутся содержательные проблемы – ведь описывается именно определенная операционная семантика (простого реляционного языка представления знаний), причем именно определенные операционные ее элементы существенны для той оптимизации времени и памяти, ради которой она задумана. Сохранив в реляционном описании лишь семантическую функцию (то есть связь знака с денотатом), выплеснем с водой и ребенка (оптимизацию ресурсов для решения конкретной задачи). Это наблюдение подтверждает ту истину, что природа знания разнообразна и различные его разновидности требуют адекватных средств. Так что и реляционный стиль, который выглядит экономным и изящным в одних случаях, может оказаться громоздким и неадекватным в других.
>
> **Вопрос.** Какие еще источники неэффективности имеются в предложенном методе поиска согласующей подстановки?
>
> **Подсказка.** Например, общий метод перебора с возвратом не защищен от многократного анализа уже проанализированных подцелей.
>
> **Вопрос.** Может ли развертка оказаться эффективнее перебора с возвратом?

4.4. О предопределенных отношениях

Внимательный читатель, по-видимому, заметил неестественность отношений, вводимых правилами **(п1)**–**(п3)** со стр. 342. Ведь по таким правилам несложно оказаться собственным братом или собственной сестрой. Конечно, следовало бы в каждое из упомянутых правил дописать условие, например (не_равно, X, Y).

Однако адекватно определить такое отношение в рамках простого реляционного языка не удастся. Конечно, в принципе можно перечислить нужные факты, касающиеся конкретного набора атомов (хотя и это занятие не из приятных – ведь нужно указать все возможные упорядоченные пары различных атомов!). Могут помочь правила, учитывающие симметричность и транзитивность отношения «не_равно». Но это не избавит от необходимости при добавлении каждого нового атома добавлять и «базовые» факты о его неравенстве всем остальным атомам.

Дело в том, что простейший реляционный язык не содержит никаких средств, позволяющих построить *отрицание* некоторого утверждения, – отрицательный ответ на запрос представлен пустым множеством положительных ответов на него.

Пример с отношением неравенства в простейшем реляционном языке показателен в том смысле, что *помогает понять фундаментальные причины появления в ЯП предопределенных* (встроенных) средств. Так как неравенство нельзя адекватно выразить средствами языка, приходится обходиться без явного определения такого отношения, считая его предопределенным (то есть фактически определенным иными средствами, выходящими за рамки ЯП).

Конечно, в реальных ЯП не для всех предопределенных средств нельзя написать явные определения.

Вопрос. Какие еще соображения могут повлиять на перечень предопределенных средств?

Подсказка. Хорошо ли выражать умножение через сложение?

Мы рассмотрели лишь простейшие примеры «программирования» в реляционном стиле. Из реальных ЯП, в которых этот стиль взят за основу, отметим отечественный Реляп [25] и широко известный Пролог [26]. В последнем, правда, имеются встроенные средства, которые лишь называются отношениями, а на самом деле служат для программирования во вполне операционном стиле.

Интересные результаты, способствующие применению Пролога в реляционном стиле, получены А. Я. Диковским.

4.5. Связь с моделями МТ и Б

Интересно проследить связь реляционного программирования с ранее рассмотренными моделями ЯП (заметим, что слово «модель» чуть выше использовалось нами в ином смысле). Стремясь к ясности, не будем бояться частично повториться в этом пункте. Зато станет прозрачнее математическая суть реляционного подхода (другими словами, будем более обычного уделять внимание математической позиции).

Итак, вернемся к исходной нашей цели – обеспечить абстракцию от программы. С позиций нашего курса можно прийти к ней разными путями. Укажем на два из них: от модели МТ и от модели Б.

4.5.1. Путь от модели Б

Основное математическое понятие в этой модели – функция из кортежей в кортежи. Программа – композиция функций.

Первая необходимая модификация на пути к модели Р – переход к более общему математическому понятию – отношению. Так как необходимо обеспечить разрешимость, рассматриваются только конечные отношения.

Определение. Конечное **отношение** – именованное конечное множество кортежей фиксированной длины. Длина кортежей называется **местностью**, или **арностью** отношения.

Для удобства положим, что имя отношения служит первой компонентой каждого его кортежа. Тогда арность – на единицу меньше длины кортежей. Например, отношение с именем «родитель» представлено совокупностью кортежей

```
(родитель, Иван, Степан)
(родитель, Иван, Дарья)
(родитель, Марья, Степан)
(родитель, Петр, Иван)
```

Содержательно это может означать, что Иван – родитель Степана, Петр – родитель Ивана и т. д.

Вторая необходимая модификация. Вводится понятие БД.

Определение. Реляционная **база данных** – это конечная совокупность конечных отношений. (От англ. *relation* – отношение.)

Третья необходимая модификация. Вместо конкретных кортежей – образцы с переменными и понятие согласующей подстановки (в точности как в модели МТ). Появляется возможность записать, например:

```
(родитель, X, Степан)
(родитель, Марья, У).
```

Уже эта модификация позволяет легко задавать вопросы (ставить задачи) относительно БД. Каждый образец можно считать вопросом о содержимом БД (а именно о совокупности согласующихся с ним кортежей). Например, наш первый образец означает вопрос «Кто родитель Степана?», а второй – «Чей родитель Марья?». В нашей БД из четырех кортежей ответы будут соответственно

```
(родитель, Иван, Степан)  и
(родитель, Марья, Степан).
```

В сущности, если у нас есть общий алгоритм поиска согласования (а он есть – ведь база конечная!), то мы достигли абстракции от программы, если считать задачей вопрос, а моделью – БД. Мы скоро увидим, что это совершенно естественно.

Обратите внимание, сколь много значит **удобная и мощная система обозначений** (какие разнообразные вопросы можно задавать с помощью *переменных в образцах*). Скачок к простоте обозначений сравним с переходом от «арифметических» формулировок задач к алгебраическим.

Четвертая модификация. Образцы становятся условными.

Определение. Условным образцом называется конечная последовательность образцов. Первый из них называется следствием, а остальные – условиями.

Например, если бы в нашей БД были отношения

```
(мужчина, Иван)
(мужчина, Степан)
(мужчина, Петр)
```

и

```
(женщина, Марья)
(женщина, Дарья),
```

то можно было бы задать условные вопросы

```
(родитель, X, Степан) (женщина, X)    и
(родитель, X, У) (мужчина, X) (женщина, У).
```

Ответы были бы

```
(родитель, Марья, Степан) и
(родитель, Иван, Дарья).
```

Другими словами, второй и последующие образцы служат *фильтрами* образца-следствия. Согласующимся с условным образцом считается только такой кортеж, согласованный со следствием, для которого все фильтры истинны (то есть

фильтры можно согласовать при подстановке, согласующей этот кортеж со следствием). Например, кортеж

```
(родитель, Иван, Степан)
```

не согласуется с последним условным образцом, так как при подстановке {X -> Иван, У -> Степан) невозможно согласовать второй фильтр (женщина, У).

Для поиска согласований с условными образцами используется перебор с возвратом (backtracking), при котором последовательно проверяют возможность согласовать последовательные фильтры и при неудаче возвращаются к предыдущему фильтру с новым претендентом на согласование. Существенно используется конечность БД.

Пятая модификация. От условных образцов к правилам-предложениям. Остался всего один принципиально важный шаг до реляционного языка представления знаний. Условный образец можно рассматривать как **правило формирования базы данных**.

Именно если существует согласующая подстановка, при которой все условия образца истинны, а следствие ложно, то условный образец можно трактовать как приказ записать в базу данных кортеж, получаемый из следствия этой подстановкой.

Конечно, в системе программирования должны быть средства, позволяющие различать трактовки условного образца как вопроса и как правила. Скажем, можно выделять правила спереди восклицательным знаком, а вопросы – вопросительным. Например, правило

```
! (дед, X, У) (родитель, X, Z) (родитель; Z, У) (мужчина, X)
```

порождает в нашей базе новое отношение

```
(дед, Петр, Степан)
(дед, Петр, Дарья).
```

Теперь мы полностью готовы к объяснению абстракции от программы в реляционном программировании.

Теория представляет собой соединение исходной БД (фактов) и БЗ – конечной совокупности правил (их называют правилами вывода, правилами порождения, хорновскими формулами, логическими соотношениями и т. п.). Описание **модели** представляет собой дополнительные факты и правила. В режиме порождения модели все правила порождения применяются для построения пополненной базы, которая и считается построенной моделью. Теперь можно ставить **задачи** на этой модели, задавая конкретные вопросы. Никакого программирования в обычном смысле не требуется – во всех случаях работают единые алгоритмы согласования образцов с кортежами.

Эта простая схема в практических реляционных языках (например, в Прологе) модернизируется для более рационального расходования ресурсов и удобства представления знаний.

4.5.2. Путь от модели МТ

Опишем его короче, учитывая сказанное выше. **Первая модификация** – полем зрения служит вся БД, при этом отношения и кортежи представлены МТ-выражениями специального вида. **Вторая модификация** – МТ-предложения превра-

щаются в правила порождения (левая часть – в последовательность фильтров, то есть правила анализа; правая часть – в следствия, то есть правила синтеза). **Третья модификация** – отменяется последовательный перебор правил – они работают все сразу и не над ведущим термом, а над всем модифицированным «полем зрения» – базой данных. Вот и все.

Итак, *реляционный стиль программирования позволяет решать задачи на новом уровне разделения труда между человеком и компьютером*.

- Человек описывает мир (представляет знания о некоторой предметной области в базе знаний).
- Человек ставит задачу (формулирует запрос к базе знаний).
- Компьютер самостоятельно решает задачу, используя известные ему факты и соотношения (правила вывода).

Можно сказать и так, что человек создает и представляет в БЗ теорию предметной области (как мы создали «теорию родственных отношений»). Затем (обычно другой человек) формулирует теорему (существования решения некоторой содержательной задачи, например теорему существования дяди у Кузьмы). Наконец, компьютер доказывает эту теорему, предъявляя решение задачи (то есть Степана в случае нашей задачи).

В связи с такой терминологией реляционное программирование называют часто **логическим**. Логическая терминология оправдана также тем, что в этой области действительно применяют методы доказательства теорем, в частности *метод резолюций*. Его характерная особенность: при подборе сопоставимого следствия выбирается наиболее общая согласующая подстановка.

С другой стороны, реляционное программирование обеспечивает абстракцию от программы, требуя от пользователя БЗ лишь постановки задачи (запроса). Такая абстракция полезна, например, для асинхронной (параллельной) реализации реляционного языка. Скажем, в очередном цикле развертка каждого правила может выполняться совершенно независимо над БЗ, используемой только для чтения.

Нетрудно предвидеть развитие правил до активных *процессов* из определенных классов, работающих над единым представлением знаний о мире. Такие процессы естественно трактовать как объекты, обменивающиеся *сообщениями* друг с другом (например, чтобы поручить решение подзадачи) и, в частности, с БЗ (например, чтобы узнать некоторый факт). Остается добавить концепцию *наследования*, и получим мостик к *объектно-ориентированному* программированию.

Еще одно перспективное направление, которое интенсивно развивается, – современные «языки представления знаний». Заинтересованного читателя отсылаем к литературе [27, 28].

Глава 5

Параллельное программирование в Оккаме-2 (модель О)

5.1. Принципы параллелизма в Оккаме 354
5.2. Первые примеры применения каналов 355
5.3. Сортировка конвейером фильтров 356
5.4. Параллельное преобразование координат (умножение вектора на матрицу) 357
5.5. Монитор Хансена-Хоара на Оккаме-2 362
5.6. Сортировка деревом исполнителей 363
5.7. Завершение работы коллектива процессов 366
5.8. Сопоставление концепций параллелизма в Оккаме и в Аде 368
5.9. Перечень неформальных теорем о параллелизме в Аде и Оккаме 377
5.10. Единая модель временных расчетов 378
5.11. Моделирование каналов средствами Ады 378
5.12. Отступление о задачных и подпрограммных (процедурных) типах 381

5.1. Принципы параллелизма в Оккаме

Одно из наиболее значительных достижений последнего времени в области информатики – появление процессоров фирмы Инмос, специально предназначенных для объединения в высокопроизводительные вычислительные среды и получивших название транспьютеров. Характерно (а для нас особенно интересно), что одновременно с транспьютерами фирма разработала язык параллельного программирования (ЯПП) и его реализацию, обеспечивающую возможность абстрагироваться от реального набора доступных процессоров. Другими словами, если отвлечься от скорости исполнения, то работа программы не зависит от количества транспьютеров (его можно учесть при трансляции программы – транслятор выполнит подходящую конкретизацию).

Важнейшая цель такого ЯПП – преодолеть представление о том, что программировать асинхронные процессы исключительно сложно. Иначе трудно надеяться на коммерческий успех транспьютеров.

Поставленной цели удалось добиться построением языка на основе принципов, предложенных одним из самых ясно мыслящих теоретиков информатики Тони Хоаром. Язык получил название «Оккам» в честь средневекового философа Уильяма Оккама (ок. 1285–1349), известного, в частности, так называемой **бритвой Оккама**. Этот принцип логических построений можно сформулировать так: *делай как можно проще, но не более того*. Бритва Оккама отсекает все лишнее в ЯП так же успешно, как принцип чемоданчика.

И если Оберон Вирта можно рассматривать в качестве минимального современного языка последовательного программирования, то Оккам, созданный под руководством Хоара, можно считать минимальным современным ЯПП. Этим он для нас и интересен.

Как обычно в этой книге, представим модель Оккама (модель О), подчеркивая ключевые концепции и принципы. Основной источник сведений – фирменное руководство по программированию на ЯПП Оккам-2 и сообщение о ЯПП Оккам-2 [31].

Поскольку параллелизмом мы уже занимались, перейдем сразу к особенностям ЯПП Оккам-2. Сначала сформулируем эти особенности, а затем проиллюстрируем их примерами.

1. Программа – это статически определенная иерархия процессов.
2. Процесс – это элементарный процесс или структура процессов. Допустимы, в частности, последовательные, параллельные, условные и иные структуры процессов.
3. В каждой структуре четко распределены роли как самих компонент, так и средств их взаимодействия. Единый принцип этого распределения таков: действия представлены процессами, память – переменными, обмен – каналами.
4. Общие переменные допустимы только у компонент некоторой последовательной структуры. (Общие константы допустимы и у компонент из различных параллельных структур.)

5. Канал Оккама – это простейший вариант асимметричного рандеву, дополненный усовершенствованными средствами связывания партнеров. Но можно считать канал и посредником между партнерами по симметричному рандеву. Такая двойственность зависит от того, рассматривается ли канал с точки зрения одного процесса (асимметрия) или пары процессов-партнеров (симметрия). Подчеркнем, что канал не обладает какой-либо памятью (обмен через канал возможен только при взаимной готовности партнеров-процессов к такому обмену).
6. Каждый канал в программе обслуживает только одну пару партнеров по обмену сообщениями. Связывание канала с парой партнеров происходит статически. При этом один партнер определяется как источник сообщений, а второй – как приемник. Партнерами по обмену могут быть только различные компоненты некоторой параллельной (асинхронной) структуры процессов. Другими словами, партнерами по обмену могут быть только параллельные процессы.
7. С каждым каналом связывается тип передаваемых по нему сообщений (так называемый протокол). Подразумевается квазистатический контроль согласованности сообщений с протоколом.
8. Коллективы (структуры) взаимодействующих процессов создаются статически посредством так называемых «репликаторов» и коммутируются посредством массивов каналов.
9. Связывание процессов с аппаратурой отделено от описания логики их взаимодействия. Так что логическая функция программы не зависит от конфигурации (размещения процессов в процессорах).

5.2. Первые примеры применения каналов

1. Буфер-переменная.

```
byte лок: -- байтовая.переменная "лок".
seq       --  последовательный комбинатор "seq"
источник ? лок -- получить из канала "источник" в "лок"
приемник ! лок -- передать из "лок" в канал "приемник"
```

2. Простейший фильтр – процесс (процедура) с двумя параметрами-каналами, протокол которых предусматривает передачу байтов.

```
proc фильтр(chan of byte источник, приемник)
   while true   — бесконечный цикл
      byte лок:
      seq
         источник ? лок
         приемник ! лок.

proc поставщик(chan of byte выход)
   while true
```

```
      byte X:
      seq
         выработать(X)
         выход ! X

proc потребитель(chan of byte вход)
   while true
      byte X:
      seq
         вход ? X
         потребить(X)
```

3. Можно теперь организовать из этих процессов параллельную структуру и связать их напрямую через канал:

```
chan of byte связь:    -- объявление байтового канала «связь»
par                    --  параллельный комбинатор «par»
   поставщик(связь)    -- компоненты такой структуры работают
   потребитель(связь)  -- параллельно (асинхронно)
```

   А можно связать те же процессы и через буфер посредством двух каналов:

```
chan of byte  связь 1, связь2:
par
   поставщик(связь1)
   фильтр(связь 1, связь2)
   потребитель(связь2)
```

4. Можно организовать буфер любого нужного размера без особого труда.

```
val int N is 50:          -- в буфере будет 50 ячеек
[N+l] chan of byte связь: -- массив из N каналов
par
   поставщик(связь [0])
   par i = 0 for N                   -- комбинатор с репликатором
      фильтр(связь [i], связь[i +1]  -- работает N фильтров
   потребитель(связь[N])             -- (i от 0 до N–1)
```

   Буфер составляется из N процессов-фильтров, способных принять очередной байт тогда и только тогда, когда предыдущий передан дальше. В результате в таком «живом» буфере каждый байт автоматически проталкивается вперед, если впереди есть место. И никаких забот о переполнении или исчерпании буфера!

Итак, благодаря каналам-посредникам с их встроенной синхронизацией-обменом-без-очередей удается работать с процессами с полной абстракцией от их асинхронной природы – они легко комбинируются в сложные структуры.

5.3. Сортировка конвейером фильтров

Интересно, что описываемая техника программирования асинхронных процессов позволяет легко превратить цепь простейших фильтров (буфер) в цепь, сортирующую поток из заданного количества чисел (например, по возрастанию).

Идея состоит в том, чтобы каждый элемент цепи выделял минимальное из проходящих через него чисел (и затем посылал его вслед прошедшему потоку). Ясно, что цепь из N таких элементов способна отсортировать последовательность из N чисел.

Напишем элемент цепи – процесс фильтр.мин.

```
proc фильтр.мин(chan of INT источник, приемник) =
   INT мин, след:
   seq
      источник ? мин
      seg i = 0 for N-1 – комбинатор с репликатором
         источник ? след
         IF
            мин <= след
               приемник ! след
            мин > след
               seg
                     приемник ! мин
                     мин := след
   приемник ! мин
```

Если заменить фильтр в предыдущей программе на фильтр.мин, получим требуемую сортировку. (Заметим, что каждый ее элемент пропускает через себя всю последовательность чисел, но общее время сортировки линейно, так как она работает по конвейерному принципу.)

5.4. Параллельное преобразование координат (умножение вектора на матрицу)

При работе с графическими устройствами часто требуется выполнять линейные преобразования n-мерного пространства по формуле

$y = Ax + b,$

где A – квадратная матрица размера n*n, а y,x,b – n-мерные векторы (x – исходный, b – постоянное смещение начала координат, y – результат преобразования).

Такие преобразования требуются, например, при отображении на экране передвижения трехмерных объектов. Поскольку нужно преобразовывать координаты каждой точки изображения, то в общем случае требуются сотни тысяч преобразований в секунду (если желательно создать эффект кинофильма или, по крайней мере, не раздражать пользователя).

Будем рассматривать трехмерное пространство (n = 3), нумеруя координаты с нуля (чтобы сразу учесть особенность индексации массивов в Оккаме). **Аналогичная задача рассмотрена в различных книгах по Оккаму, например в [29], но там Оккам старый.**

Итак, наша программа должна вычислить координаты вектора y по формуле

(*) y[i] = SIGMA(a[i,j] * x[j]) + b[i],

где SIGMA – сумма по j от 0 до 2 (то есть n – 1).

Если основные затраты времени приходятся на умножения и сложения, а время дорого, то имеет смысл обратить внимание на возможность обеспечить распараллеливание всех нужных умножений (их ровно n**2 – в нашем случае 9) и некоторых сложений.

Если бы удалось поручить каждое умножение отдельному исполнителю (процессору), то можно было бы ускорить вычисления (в принципе) в n**2 раз! На абстрактном уровне ЯПП исполнители представлены процессами, и при наличии физических исполнителей компилятор позаботится сам о размещении различных процессов на разных физических процессорах (или учтет явные пожелания программиста). Во всяком случае, именно так работает компилятор Оккама-2.

Однако, чтобы распределить умножения, требуется, во-первых, передать каждому процессу его аргументы; во-вторых, уложиться в ограничения используемого ЯПП (в нашем случае главное из них – то, что канал может связывать точно два процесса). Наконец, в-третьих, нужно не только умножать, но и складывать, а также «поставлять» аргументы и «забирать» результаты.

5.4.1. Структура коллектива процессов

Из формулы (*) видно, что каждому элементу a[i,j] матрицы A естественно сопоставить отдельный процесс pa[i,j], выполняющий умножение на этот элемент значения x[j].

Если принять за основу эту идею (обеспечивающую требуемое ускорение в n**2 раз), то остается вторая ключевая проблема – обеспечить подходящую коммутацию процессов pa[i,j]. Нужно ввести подходящие процессы-партнеры и связать их подходящими каналами (по одному на пару партнеров!).

Понятно, что все pa [i,j] должны рано или поздно (лучше раньше) получить по каналам значения x[j] и b[i]. Дисциплина «один канал на пару партнеров» требует создать по процессу px[j] на каждый элемент вектора x и по процессу pb[i] на каждый элемент вектора b[i].

С другой стороны, эта же дисциплина не позволяет передавать x[j] сразу нескольким pa[i,j] по одному и тому же каналу. Естественно желать, чтобы число каналов, связанных с процессом, было фиксировано (это упрощает и его представление на физическом уровне). Поэтому приходится применить сквозную передачу данных через pa[i,j]. Этот характерный для Оккама элемент стиля программирования можно назвать **технологией фильтров**. В нашем случае фильтрами служат pa[i,j].

Замечание (о фильтрах). В технологии фильтров каждый процесс рассматривается как некоторый преобразователь потока данных (действие которого полностью сводится к преобразованию потока). Название «фильтр» связано с тем, что в проходящем через преобразователь потоке подвергается обработке только вполне оп-

ределенный класс данных – остальные передаются в выходной поток без изменений. Обработанные фильтром (свои) данные заменяются в выходном потоке результатами обработки без нарушения естественного порядка в исходном потоке.
Технология фильтров помогает «нанизывать процессы на потоки данных» как бусинки, собирая из относительно элементарных бусинок разнообразные программы-сети. Полная аналогия с электрическими сетями, собираемыми из стандартных элементов.
Важны отсутствие побочных эффектов и простота коммутации. Сравните с подключением процедуры, где нужно готовить аргументы, хранить результаты, беспокоиться о глобальных ресурсах. Впрочем, аналогичная технология успешно применяется и в последовательном программировании (на Фортране, Паскале, Форте и др.) на основе процедур-фильтров. Этот стиль используется в командном языке ОС UNIX и в программировании на ЯП Си.

Наконец, следует заготовить процессы py[i], получающие компоненты результирующего вектора y[i], а также процессы p0[j], назначение которых станет ясным чуть позже. Получается следующая схема (рис. 5.1):

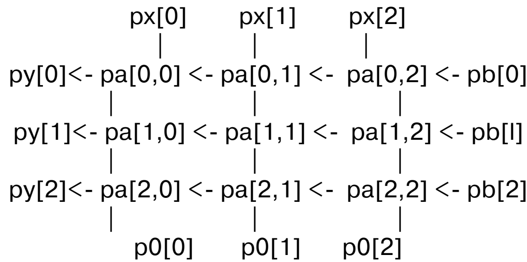


Рис. 5.1

Таким образом, конфигурация процессов разработана. Стрелки указывают направление потоков данных (им и должны соответствовать каналы).

Конечно, можно было бы для нашего случая описать каждый процесс (всего их 21) в отдельности. Однако это громоздко, и к тому же упустим главную цель – показать Оккам-2 в действии. Поэтому опишем процесс каждого вида процедурой с параметрами (с тем чтобы затем воспользоваться средствами компоновки нужной конфигурации из таких процедур).

Начнем с процесса вида pa. Такой процесс должен воспринимать «сверху» значение x[j], справа – частичную сумму b[i] с «правыми» произведениями (до a[i,j-l] * x[j-1] включительно) и передавать вниз без изменений значение x[j], а налево – модифицированное на a[i,j] * x[j] значение частичной суммы. Параметрами процесса pa, следовательно, должны быть значения a[i,j] и четыре канала, два из которых – для входных потоков, два – для выходных. Итак,

```
proc pa(val real aij, chan real32 верх, низ, лево, право) =
eal xj,yi,aij.xj :
seg
```

```
  верх?xj
  while true                -- аналог loop
    seg
      par
        низ!xj              -- передать xj
        aij.xj := aij * xj
        право?yi            -- запрос частичной суммы
      par
        лево!yi + aij.xj
        верх?xj
```

В результате повторений цикла можно обрабатывать при необходимости много векторов (в зависимости от того, сколько поступит). Напишем параметрический процесс pb. Он служит источником значений вектора b[i].

```
proc pb(val real32 bi, chan of real лево) =
  while true
  лево ! bi
```

Вопрос. Зачем здесь бесконечный цикл?

Подсказка. Это единственный способ обеспечить значениями работающие в аналогичном цикле процессы pa[i,j].

Процесс px поставляет значения вектора x[j].

```
proc px(val real j, chan of real низ) =
  ... -- создать x(j) каким-то способом в зависимости от j
  while true
    низ ! создать.x(j)
```

Процесс py получает значения вектора y[i]

```
proc py(chan of real право) =
  real куда.то:
  while true
  ...
  право ? куда.то
  ...
```

Ясно, что два последних процесса – своего рода заглушки. Они обязательно нужны, чтобы у внутренних процессов были партнеры.

Вопрос. Что произойдет при отсутствии или остановке таких партнеров?

Содержательно эти два процесса могут представлять собой, например, связь с внешним миром, для описания которой в Оккаме имеются специальные средства (например, так называемые **порты**).

Пора вспомнить о процессах p0. Они нужны только затем, чтобы в нижней строке нашей структуры процессов могли стоять процессы вида pa (то есть с каналами «вниз»), хотя здесь уже вниз ничего передавать не надо. Процессы вида p0 также играют роль заглушек – они просто поглощают (содержательно лишние) передачи вниз. Можно считать, что вместе с нижним рядом процессов вида pa они должны образовать специальные процессы вида pal – без передачи вниз.

```
proc p0(chan of real верх) =
   real куда.то:
   while true
   ...
   верх ? куда.то
```

5.4.2. Коммутация каналов

Все процессы описаны. Осталось самое неприятное при программировании на Оккаме – обеспечить их правильное «размножение» и правильную коммутацию каналов. Первое делается за счет так называемых **репликаторов** (размножителей), второе – за счет массивов каналов, связываемых затем покомпонентно с каждым экземпляром процесса нужного вида.

Сначала напишем, потом прокомментируем.

```
val int n is 3
[n] [n] real a   -- массив a[i,j], индексы с нуля до n–1
[n] real b       -- массив b[i]
seq
   ... -- присваивание значений компонентам a и b
   [n+1] [n] chan of real верх.низ          -- эти массивы локальны
   [n] [n+1] chan of real право.лево        -- в ближайшем «par»
   par
      par j=0 for n– max j=n–l!
   px(j, верх.низ[0] [j])                   -- самые верхние каналы
      par i=0 for n
         pb(b[i], право.лево [i] [n])       -- правые каналы
      par i=0 for n
         par j=0 for n
            pa(a[i] [j], верх.низ[i] [j],   -- верхние для (i,j)
                    верх.низ [i+1 ] [j],    -- нижние
                    право.лево [i] [j],     -- левые
                    право.лево [i] [j+1])   -- правые
      par j=0 for n– max j=n-l
         p0(верх.низ[n] [j])                -- самые нижние
      par i=0 for n
         py(право,лево [i] [0])             -- самые левые
```

Вопрос. Не заметили ли вы нарушение принципа целостности по отношению к массивам a и b?

Подсказка. Сколько раз пришлось указывать параметр n?

Итак, объявлены и инициализированы массивы постоянных, специфичных для конкретного запуска программы (массивы a и b). Затем объявлены подходящих размеров двумерные массивы каналов. Следует учесть, что нумерация индексов в массивах Оккама – с нуля.

Достаточно взглянуть на структуру наших процессов (стр. 359), чтобы убедиться в правильности структуры массивов каналов – двенадцать стрелок-кана-

лов сверху вниз и двенадцать стрелок справа налево. Объявленные массивы каналов локальны в ближайшем непосредственно следующем процессе (начинающемся с открывающей скобки-комбинатора par).

Ключевой момент – согласовать репликацию (размножение) процессов со связыванием их аргументами-каналами. Партнерам по обмену требуется передать нужный канал, причем в соответствии с ролями партнеров: одному – в качестве входного канала-аргумента, второму – в качестве выходного.

Легко видеть, что именно так и делается. Например, самый нижний pb[2] получил в качестве выходного канала право.лево [2] [3]. Этот же канал получил в качестве входного справа процессора pa [2] [2], что и требовалось.

Обратите внимание, что все размноженные процессы (всего их 21) работают параллельно, хотя при их коммутации можно было совершенно не думать о параллелизме.

> **Упражнение.** Постарайтесь найти ошибки в приведенной программе на Оккаме-2 (или обоснуйте ее правильность).

5.5. Монитор Хансена-Хоара на Оккаме-2

```
proc буф(chan of byte связь 1, связь2)
[n] byte буфер:
seq
   proc занести(val byte x)
   ...                        : -- конец объявления "занести"
   proc выбрать(byte x)
   ...                        :
   boot function полон
   ...                        :
   bool function пуст
   ...                        :
   byte z
   while true                 -- полный аналог loop в Аде
      alt                     -- аналог select в Аде
         not полон & связь1 ? z
            занести(z)
         not пуст & связь2 ! z
            выбрать(z)
         true & SKIP
```

Это полный аналог монитора в Аде. Пример полезен и тем, что позволяет познакомиться еще с одной конструкцией Оккама – оператором alt (аналогом select в Аде). В нашем примере в этом операторе три альтернативы. Аналогично select рассматриваются сначала лишь те, где имеется заказ рандеву. Если среди них найдутся открытые (то есть с истинными условиями, причем такие, где рандеву может состояться (партнер готов)), то выбирается одна из них и выполняется. Иначе выполняется всегда открытая альтернатива со SKIP.

Обратите внимание, что задержка (оператор ожидания) не используется, – активное ожидание на отдельном физическом процессоре не хуже простоя.

Если нужно побыстрее освобождать буфер, когда имеются заказы на оба рандеву (по двум каналам), то можно воспользоваться разновидностью оператора alt с заголовком pri alt. В нем высшим приоритетом обладает та альтернатива, которая расположена ближе к заголовку. Так, в нашем случае нужно было бы написать

```
pri alt
   not пуст & связь2 ! z
      выбрать(z)
   not полон & связь 1 ? z
      занести(z)
   true & SKIP.
```

Стоит подчеркнуть, что выигрыш в скорости достигается только при реальной возможности работать с коллективом физических процессоров. Иначе будем проигрывать из-за накладных расходов на моделирование параллельного исполнения.

Вопрос. Почему в Оккаме нет объявления входа?

5.6. Сортировка деревом исполнителей

Чтобы закрепить «новое параллельное мышление», продолжим серию примеров. Опишем сортировку слиянием, рассчитанную на потенциально не ограниченный массив сортируемых чисел.

Общий замысел. Коллектив исполнителей представляет собой двоичное сбалансированное дерево (как увидим, сбалансированность нужна только для идентификации каналов). Дерево исполнителей воспринимает сортируемый поток чисел через свой корень и возвращает обратно отсортированный по возрастанию поток. Предполагается, что общее количество чисел в потоке ограничено, однако коллективу неизвестно. Вместе с тем листьев в дереве достаточно для размещения всех чисел потока.

Ключевая идея. Построить дерево из однотипных процессов, каждый из которых, во-первых, распределяет входной поток между своими потомками в дереве и, во-вторых, сливает отсортированные потомками обратные потоки, направляя результат своему предку.

Детали. *Во-первых*, как и в задаче о перемножении векторов, важно удачно занумеровать каналы. Наше дерево будет иметь вид, показанный на рис. 5.2.

Из рисунка ясно, что у каждого внутреннего процесса-узла (то есть не листа и не корня) – шесть каналов (два – для связи с предком, четыре – для связи с потомками). Имеются два вида каналов – для передачи вверх и вниз по дереву.

Каналов каждого вида ровно столько, сколько исполнителей, а именно $2n - 1$, где n – число листьев в дереве. Если нумеровать их с корня, то получается, что процесс с номером i связывают с предком каналы с номером i, а с потомками – каналы с номерами $2i + 1$ и $2i + 2$. На этих расчетах и строится коммутация каналов.

3 4 5 6
1 2
0

Структура дерева

Влево Слева Вправо Справа
Узел
Снизу Вниз

Шесть каналов одного узла

Рис. 5.2

Во-вторых, нужно учесть неопределенность длины потока. Обычный прием – завести признак конца потока. Так мы и поступим, однако придется решить еще один вопрос.

Дело в том, что по многим причинам желательно добиваться однородности потока. Другими словами, его элементы должны иметь фиксированный тип. Если не ограничивать допустимые значения сортируемых чисел, то единственная возможность иметь признак конца потока – представить элемент потока объектом комбинированного типа, то есть парой, состоящей из собственно числа и признака (например, булевого) продолжения потока.

Поэтому примем, что поток состоит именно из таких пар, причем ложь в поле продолжения означает, что мы имеем дело с последним элементом потока. Точнее говоря, потоков будет много, и все они устроены аналогично.

Вопрос. Откуда возьмется много потоков?

Подсказка. Исходный поток распределяется по ветвям дерева.

Собственно программа. Итак, нужны три вида процессов – предкорневой (драйвер, представитель внешней среды, откуда поступает исходный поток и куда отправляется отсортированный), сортировщик (рядовой процесс-узел) и лист (терминальный процесс, способный лишь получать и возвращать числа, поворачивая поток вспять). Вот как организовать из них дерево:

```
val INT колич.чисел is 250 :    -- константы-параметры потока
val INT глубина is 8 :          -- и дерева
val INT число.листьев    is 1 <<глубина, -- 2**глубина = 256
   INT число.узлов       is число.листьев–1,
   INT число.процессов is число.листьев + число.узлов,
   INT число.каналов is число.процессов,
   INT корень is 0,
   INT первый.узел is корень,
   INT первый.лист   is первый.узел + число.узлов :
PROTOCOL пары is bool; real :  -- именованный протокол «пары»
      -- определяет структуру последовательностей, передаваемых по тем каналам,
      --  при спецификации которых указан такой протокол
[число.каналов] chan of пары верх, низ:   -- два массива каналов
   ...                                    -- объявления нужных процессов
```

```
par
   драйвер(верх [корень], низ [корень])
   par i= первый.узел for число.улов
      узел(верх [i], низ[i], верх[2*i+l], низ[2*i+1], верх [2*i+2], низ [2*i+2])
   par i = первый.лист for число.листьев
      лист(верх [i], низ [i])
```

Тем самым структура процессов задана. Осталось реализовать общий замысел, сосредоточившись на каждом виде процессов. При этом каналы позволяют полностью отвлечься от асинхронной природы процессов.

Объявления процессов. Основной процесс-сортировщик (узел) легко представить двумя параллельными процессами. Первый распределяет исходный поток по потомкам, второй сортирует слиянием обратные потоки. Будем считать, что поток поступает снизу вверх, так что предок находится внизу, а потомки – сверху.

```
proc распред(chan of пары низ, лево, право) =
   -- по каналам-параметрам "низ", "лево", "право" передаются
   -- пары (bool; real), в соответствии с протоколом "пары"
   val bool влево is true,
      bool вправо is false :
   bool направление, еще      :
   seq
      направление := влево
      низ ? еще
      while еще                           -- цикл до конца потока
         real число :
         seq
            низ ? число
            if направление = влево
                  лево ! true; число      -- выдать пару
               направление = вправо
                  право ! true; число
            низ ? еще
            направление := not направление  -- смена направления
   par
      лево ! false                        -- признак конца потока
      право ! false:
```

Комментарии, по-видимому, излишни.

```
proc слияние(chan of пары низ, лево, право) =
   bool лево.еще, право.еще :
   real левый.мин, правый.мин :         -- минимумы
   seq
         par
            лево ? лево.еще; левый.мин   -- ввод парами
            право ? право.еще; правый.мин
      while лево.еще or право.еще
            if лево.еще and              -- так можно прерывать строчку
            ((not право.еще or(левый.мин < правый.мин))
               seq
```

```
                  низ ! true; левый.мин
                  лево ? лево.еще; левый.мин
                     -- ведь предыдущий не последний
               право.еще and            -- вторая альтернатива if
               ((not лево.еще or(правый.мин < левый.мин))
               seq
                  низ ! true; правый.мин
                  право ? право.еще; правый.мин
   низ ! false; any — "заполнитель" для нормального конца (см. ввод по каналу "низ")

proc узел(chan of пары снизу, вниз, влево, слева, вправо, справа) =
   par
      распред(снизу, влево, вправо)         -- взаимодействие
      слияние(вниз, слева, справа) :        -- через потомков
proc драйвер(chan of пары верх, низ) =
      real число :
         seq
            seq i = 0 for колич.чисел
               seq
                  число := создание.чисел  -- какая-то процедура
                  верх ! true; число
            верх ! false; any        -- посылка "звонка" о конце работы
                                     -- согласовано с приемом в процессе "узел"
         seq
            seq i = 0 for колич.чисел
               seq
                     низ ? any; число   -- чтобы "съедать" булевы
                     обработать.(число) -- обработка чисел
            низ ? any; any              -- "съедание" признака конца
                                        -- (получение звонка об окончании работы)

proc лист(chan of пары верх, низ) =
real число :
seq
   верх ? any; число               -- прием пары
   низ ! true; число; false; any   -- поворот потока  и посылка звонка
   верх ? any                      -- прием звонка и сразу конец
```

Лист работает ровно один раз, поэтому прием звонка в нем нужен только для того, чтобы не мешать работе связанного с ним узла (см. ниже).

5.7. Завершение работы коллектива процессов

Суть проблемы – в том, что из-за связи по каналам, требующим взаимной готовности процессов, члены коллектива оказываются весьма чувствительными к любому нарушению нормального режима работы. В частности, неосторожное завер-

шение работы некоторого члена коллектива (например, в момент, когда кажется, что вся возложенная на него работа закончена) легко может привести к «зависанию» членов коллектива, ожидающих несостоявшихся рандеву.

Например, представим себе, что драйвер (подходящим образом модифицированный) обнаружил в потоке недопустимый объект. Ему нельзя просто выдать диагностику и завершить работу. Ведь взаимодействующие с ним процессы-узлы будут ждать нужных им рандеву. А так как ждать станет некого, возникнет тупиковая ситуация (которая может коснуться неограниченного количества процессов). При этом на выходе драйвера (по каналу «низ») в общем случае не будет получена даже та часть потока, которую удалось нормально отсортировать (так как она «застряла» в процессах-узлах).

Основной тезис таков: следует считать, что *на каждого члена коллектива асинхронно работающих процессов возложена забота об аккуратном информировании партнеров о предстоящем завершении работы*. Это естественная *плата за параллелизм* (и простоту каналов как средств взаимодействия).

> **Упражнение.** Придумайте механизм взаимодействия, снимающий заботу о корректном завершении работы с каждого партнера.
>
> **Подсказка.** Конечно, такой механизм должен использовать специальный сигнал завершения работы, «понимаемый» всеми партнерами.

Опишем **вариант правильной стратегии поведения процессов**. Процесс, решивший закончить свою работу, должен:

1) передать потенциальным партнерам (то есть партнерам, с которыми еще возможно содержательное взаимодействие) сообщение со смыслом «заканчиваю»;
2) получить от всех потенциальных партнеров сообщение со смыслом «услышан» (это может быть, например, также сообщение «заканчиваю», но полученное от процесса, который перестает быть потенциальным партнером после передачи такого сообщения);
3) не заказывать рандеву с процессами, приславшими сигнал «услышан»;
4) закончить работу.

Таким образом, сообщение «заканчиваю» дает возможность партнерам подготовиться к отсутствию рандеву в будущем, а сигнал «услышан» дает возможность инициатору завершения работы понять, что рандеву с соответствующим процессом больше не будет.

Конечно, в некоторых случаях эту стратегию можно упростить. Например, если в процессе нет приема, то ему можно завершать работу сразу после своего «заканчиваю», а если в нем нет передачи, то можно заканчивать после приема «заканчиваю» от всех потенциальных партнеров.

В нашей сортировке сообщение «заканчиваю» представлено передачей «false». Драйвер, выступая инициатором завершения, посылает вверх «false», но не заканчивает получение и передачу вниз отсортированной последовательности, пока не получит «false» сверху. Процесс-узел, со своей стороны, сначала посылает сообщения «заканчиваю» своим потомкам, а завершает работу лишь после получения «false» от потомков и его передачи предку.

Несколько дополнительных вопросов

1. ***Почему в драйвере внешний комбинатор «seq»?*** Что изменится, если поставить «par»?

Логически ничего не изменится, так как второй seq сможет что-либо получить по каналу «низ» только после посылки «false» по каналу «верх». Однако указанная замена недопустима по формальным соображениям.

Дело в том, что переменная «число» формально станет разделяемой между двумя параллельными процессами (хотя, как сказано выше, фактически второй процесс получит доступ к ней строго после первого). Это противоречит принципам Оккама и вызовет соответствующую диагностику компилятора.

2. ***Можно ли первый вложенный «seq» в драйвере заменить на «par»?***

Нельзя из-за разделяемой переменной «число».

А если ее объявление поставить перед самым внутренним комбинатором «seq» (то есть сделать ее локальной внутри следующей конструкции)?

Теперь переменная «число» перестает быть разделяемой между параллельными процессами. Однако остается нарушенным еще один принцип Оккама: один канал «верх» не может обслуживать более двух асинхронных партнеров! Остается и содержательное возражение – потребуется синхронизация запусков процедуры создание.чисел (**зачем?**).

Итак, каналы Оккама отличаются от симметричного рандеву только тем, что каждый канал статически закрепляется за вполне определенной парой партнеров. Модификация семантики небольшая, однако становится существенно проще понимать и контролировать конфигурацию процессов.

5.8. Сопоставление концепций параллелизма в Оккаме и в Аде

Основное отличие концепции параллелизма, принятой в Аде, от концепции Оккама состоит в том, что если в Оккаме асинхронный процесс – обычная (рядовая) компонента программы, то в Аде каждый асинхронный процесс – объект, требующий специального объявления.

Другими словами, в Оккаме описания процессов мыслятся в первую очередь как компоненты иерархии действий, а в Аде – как компоненты иерархии объектов определенных (а именно задачных) типов. Специфика задачных типов и определяет действия (операции), применяемые к объектам этих типов.

В конечном итоге и в Оккаме (как было видно) можно объявлять именованные процессы, и в Аде – управлять запуском, взаимодействием и завершением процессов. Однако исходная концепция существенно влияет на выбранные для этого авторами выразительные средства.

Постараемся показать это на примере описания на Аде коллектива процессов, преобразующих координаты векторов. Главная наша цель – продемонстрировать отличия Ады от Оккама в управлении асинхронными процессами. Для начала

уточним изложенную ранее концепцию параллелизма в Аде, а затем перейдем к программированию.

5.8.1. Концепция параллелизма в Аде

Итак, прежде чем иметь дело с асинхронным процессом в Аде, его нужно определить посредством объявления задачи. Примером может служить объявление задачи **«буф»**, приведенное в качестве описания монитора Хансена-Хоара. Оно состоит из спецификации и тела задачи. Спецификация содержит только объявления входов, то есть названия видов рандеву (процедур-рандеву), предоставляемых в качестве услуг клиентам процесса «буф». Тем самым объявляемый процесс с точки зрения этих видов рандеву становится мастером (обслуживающим процессом), что не мешает ему выступать клиентом в других видах рандеву, если в его теле вызываются другие процессы-мастера.

Соответствующий объявлению задачи процесс запускается в результате так называемого **предвыполнения** (обработки) этого объявления и выполняется асинхронно с запустившим его процессом (среди объявлений которого находится рассматриваемое объявление задачи). Запустивший процесс называется **предком**, а запущенный – его **потомком**. Точнее говоря, одновременно запускаются все непосредственные потомки процесса-предка.

В Аде предусмотрены исключения, возникающие при попытках заказать рандеву с еще не запущенным или уже завершенным процессом. Правило одновременного запуска потомков гарантирует им возможность заказывать рандеву между собой, не опасаясь попадания в аварийные ситуации.

Процесс завершается, когда управление достигает конца его тела или когда выполнится оператор terminate (внутри процесса) или abort (вне процесса). Предварительно завершаются все потомки процесса.

Объявления коллективов процессов. Объявление задачи «буф» представляет лишь один вид объявления. В этом случае формально считается объявленным анонимный задачный тип, единственным объектом которого и считается объявленный процесс (в нашем случае – «буф»).

Однако в общем случае можно объявить именованный задачный тип и затем объявлять отдельных его представителей обычными объявлениями объектов (этого типа). Например:

```
task type буф is    -- добавлено слово type
   entry поставить(X : in сообщение);
   entry получить(X : out сообщение);
end буф;
task body буф is
   ...             -- тело совершенно то же
end буф;
   ...
A, B : буф;        -- обычное объявление двух объектов.
```

При обработке такого объявления создаются и одновременно запускаются два процесса А и В типа «буф». Клиенты могут воспользоваться услугами «мастеров», заказывая соответствующие рандеву посредством вызовов входов

```
A.поставить(Z1); B.поставить(Z2);
B.получить(Y1); A.получить(Y2); и т. п.
```

Обратите внимание: входы задач можно считать аналогами селекторов в записях. Вызов входа – аналог выборки поля записи. Это характерная операция для задачных типов. Но в определяющем пакете для конкретного заданного типа можно объявить и иные операции (использующие в качестве элементарных вызовы входов). Например, операцию «передать слово», использующую вызов входа, работающий с одним байтом. Все это дает основания считать задачные типы полноценными (ограниченными приватными!) типами данных, причем данных *активных* (а не привычных пассивных).

Как только асинхронные процессы оказываются обычными объектами данных (в Аде), становится естественным строить из них структуры, в частности массивы и записи. Никаких новых выразительных средств для этого не требуется. Например, можно объявить тип – массив буферов

```
type масс_буф is array 1...10 of буф;
```

и конкретный массив этого типа

```
M : масс_буф;
```

а затем заказать рандеву с i-м элементом этого массива М

```
M(i).поставить(Z);
```

Применимы к задачным типам и обычные средства образования динамических структур – ссылочные типы с генератором new. Например:

```
type P is access буф;
P1 : P;
   P1 := new буф; — создание и запуск нового процесса-буфера
   P1.поставить(Z); — и т. п.
```

Итак, и в Аде имеются средства объявления коллектива исполнителей – асинхронных процессов. Их можно объявлять **поштучно** – объявлениями задачи, или объявлениями объектов задачного типа, а также **массово-статически** – объявлениями массивов и, наконец, **массово-динамически** – посредством ссылочных задачных типов.

Коммутация процессов в Аде. Однако с коммутацией процессов-исполнителей возникают проблемы, в основном связанные с отсутствием в Аде понятия канала. Короче говоря, в Аде отсутствуют естественные средства статического связывания коллектива процессов (аналогичные репликаторам и массивам каналов в Оккаме).

Действительно, входы и вызовы входов фиксированы при объявлении задачного типа. Чтобы связать два объекта этого типа, нужно передать им (или хотя бы одному из них) атрибуты партнера. Однако при объявлении типов еще нет нужных объектов, а при объявлении объектов (и массивов процессов) передать нужные атрибуты нельзя (инициализация ограниченных приватных запрещена, да и в общем

случае средства инициализации в Аде слабы для установления нужных связей – ведь каждый элемент массива должен получить «имя» партнера по рандеву).

Сравните, в Оккаме члены коллектива связывались каналами именно при описании структуры коллектива (посредством репликаторов).

Поэтому в Аде структуру коллектива процессов приходится создавать *динамически* (привлекая аппарат динамических параметров и, возможно, ссылок).

Теперь все готово для попытки воплотить в Аде параллельное преобразование координат.

5.8.2. Параллельное преобразование координат в Аде

Вернемся к примеру, рассмотренному ранее для Оккама на стр. 357. Предстоит решить несколько проблем. Покажем их на примере процессов pa.

Статическое объявление процессов

Во-первых, придется описывать коллектив процессов этого вида. Сформируем его статически, а связывать процессы будем динамически (так как статических средств для этого нет, см. выше). Ясно, что придется ввести массив процессов. Стало быть, нужно описать тип элементов такого массива. Конечно, это должен быть задачный тип Ады. Итак, нужно объявить задачный тип, например тип pa, и массив объектов этого типа.

Во-вторых, нужно понять, какие входы нужны объектам типа pa. Прототип в Оккаме имел четыре канала, и это соответствовало симметричному рандеву. Так как в Аде рандеву асимметричное, нужно решить, по каким направлениям процесс будет играть роль мастера, а по каким – клиента, и ввести нужные входы. Примем, что процесс pa предоставляет услуги партнерам сверху (для принятия xj) и партнерам справа (для принятия заготовки yi). При этом он сам пользуется услугами партнера снизу (для передачи ему xj) и партнера слева (для передачи ему модифицированного yi).

Итак, нужны два входа, которые назовем «сверху» и «справа». Запишем только что разработанный вариант фрагмента программы

```
task type pa is
   entry сверху(X : in real);   -- принять xj сверху
   entry справа(X : in real);   -- принять yi справа
end pa;
```

В теле pa должны быть и операторы приема (этих двух входов), и вызовы входов (партнеров снизу и слева). Так что тело pa имеет вид

```
task body pa is
   xj, yi,. aij_xj : real;
   ...
   accept сверху(X : in real) do   -- верх ? xj
      xj := x                      -- прием от партнера сверху
   end сверху;
```

```
    ...
  accept справа(Y : in real) do    -- право ? yi
      yi := x                      -- прием от партнера справа
  end справа;
    ...
  партнер_снизу.сверху(xj);        -- низ ? xj
      ...-- передача партнеру вниз
  партнер_слева.справа(yi);        -- лево ? yi
      ...-- передача партнеру влево
end pa;
```

Если сопоставить с текстом ра на Оккаме, сразу видно неудобство записи на Аде «малого параллелизма». Не удается выразить параллелизм внутри процесса ра без новых объявлений задач (со своими входами, параметрами, вызовами входов). Длинно, неудобно, ненаглядно – Ада для такого параллелизма не приспособлена!

В-третьих, самый существенный вопрос. Как сообщить процессу типа ра конкретные имена его партнеров (то есть процессов, с которыми следует связать имена партнер-снизу и партнер-слева)? Да и значения aij ему также требуется передать. Выше объяснено, почему это приходится делать динамически (в частности, при объявлении типа ра процессов-партнеров еще просто нет). Следовательно, при объявлении типа ра необходимо позаботиться о настройке процессов (на связь с конкретными партнерами) в период исполнения программы. Мы пока этого не сделали!

При создании массива из элементов типа ра можно было бы согласованно инициализировать его элементы, присвоив им подходящие значения. Но для этого нужно иметь возможность «вычислить» нужный процесс и «присвоить» его компоненте массива. Ада не дает возможности «вычислять» процессы нужным образом, да и присваивания ограниченным приватным объектам запрещены, а тем самым и инициализация тоже.

Итак, связать партнеров статически в Аде невозможно (если не все партнеры известны в момент создания программы).

Подчеркнем, что наши проблемы с коммутацией процессов вызваны в значительной степени тем, что средства управления рандеву в Аде нельзя отделить от процессов (точнее, от объявления задач). В Оккаме каналы – самостоятельные объекты (предназначенные для связывания других самостоятельных объектов – процессов). Именно разделение этих двух дуальных видов объектов позволяет легко отложить связывание процессов от момента описания процесса (и каналов) до момента порождения конкретного процесса. Канал становится естественным параметром процесса, причем параметром периода компиляции. Нетрудно породить столько каналов, сколько нужно для связывания, и указать подходящий канал в качестве аргумента процесса. При этом передача аргумента-канала каждому партнеру выполняется полностью независимо (при порождении партнера).

Обратите внимание: удачно работает адекватная абстракция-конкретизация (абстракция от партнеров – *канал* как абстрактное симметричное рандеву, и конкретизация – настройка канала сначала на одного партнера, затем на другого).

В Аде именно отсутствие такой абстракции приводит к рассматриваемой нами сейчас проблеме коммутации. Кстати, появление нужной абстракции вполне возможно прогнозировать на основе общих критериев качества абстракции-конкретизации – нужно лишь учесть технологическую потребность в развитой коммутации процессов.

Организация динамической настройки процессов

Итак, в Аде средств коммутации при порождении процессов нет. Приходится программировать динамическое связывание партнеров. Но динамическое – значит при исполнении программы, а единственное средство связи с работающим процессом в Аде (как и в Оккаме) – рандеву. Причем это обязательно должно быть рандеву родительского процесса со своими потомками – потомки до настройки еще не могут «знать» друг друга! **(Почему?)**

Такое настраивающее рандеву в общем случае должно передать процессу либо информацию только о его собственных координатах (с тем чтобы сам процесс «вычислил» своих партнеров), либо непосредственно о самих партнерах. Хотя первое решение в Аде воплотить проще, рассмотрим именно второй вариант по двум причинам. Во-первых, мы сопоставляем возможности Ады и Оккама по части коммутации процессов, а в исходной программе на Оккаме процесс вида ра не вычислял своих партнеров, а был настроен на них. Во-вторых, не менее существенно, что нетрудно так обобщить постановку задачи, чтобы процесс вида ра в принципе не мог вычислить не только конкретного партнера, но даже его тип.

Вопрос. Как это сделать?

Подсказка. В сущности, от процессов, например, вида ру требуется лишь способность воспринимать информацию, передаваемую процессами вида ра.

Итак, будем считать, что при настройке процессу вида ра необходимо передать информацию, позволяющую ему обратиться к партнеру в общем случае заранее неизвестного вида (типа). Мы пришли к необходимости подправить описание процесса ра – нужен еще один вход для настройки процессов. Его параметрами должны стать aij, а также партнеры снизу и слева.

Принципиальные решения приняты: массив процессов типа ра (двумерный) и три входа – два для работы, один для настройки. Однако технические проблемы остаются.

Первая техническая проблема – параметрами входа «настройка» в объектах типа ра могут оказаться объекты того же типа ра. И тип ра становится объявляемым непосредственно через себя, что в Аде недопустимо. Ведь использовать имя типа до его полного объявления можно лишь в так называемых **неполных объявлениях** (предобъявлениях) перед объявлением ссылочных типов. Приходится вводить тип ссылок на ра:

```
type pa;
type ppa is access pa;
type pa is . . .;
```

Вторая техническая проблема – для настройки необходимо вычислить ссылки на отдельные компоненты массивов процессов (если, как мы и договорились, не заставлять процесс вида pa «знать» о том, что одни его партнеры организованы в массив определенной структуры и доступ к ним возможен посредством индексов этого массива, другие партнеры – в массив другой структуры, и т. п.). В Аде есть предопределенная атрибутная функция для вычисления адреса объекта типа P – P'ADRESS, однако она доставляет не значение нужного типа (а именно ppa), а значение типа address из предопределенного пакета SYSTEM. Из-за этого придется заменить массив процессов типа pa массивом ссылок типа ppa. Сами процессы типа pa нужно будет порождать операцией new (то есть динамически).

Так что отсутствие адекватной абстракции (от партнера) привело через потребность в динамической настройке к необходимости динамического порождения процессов – все больше нагрузка на целевую машину (столь неприятная для Ады – **почему?**).

Третья техническая проблема – как учесть «краевые элементы». Ведь «нижние» и «левые» процессы типа pa должны общаться с процессами совсем другого типа.

Нам снова не хватает посредников – каналов Оккама. Для них было не важно, с процессом какого типа связать, – лишь бы протоколы были согласованы (кстати, еще одна полезная абстракция – от *типа* связываемого процесса). А в Аде мешает контроль за типами параметров у входа «настройка» (**почему?**).

Решение можно построить на основе так называемого вариантного комбинированного типа, дискриминантом которого стал бы «вид_процесса», а выбираемыми компонентами – процессы разных типов. При этом в самих этих процессах придется разбираться, процесс какого типа оказался партнером слева или снизу, и использовать подходящий селектор записи (единым селектором для процессов разных типов не обойтись – опять же из-за контроля типов). Тип процесса (точнее, его признак) придется задавать при настройке (конечно, вместе с дискриминантом вариантной записи).

Рассмотрим это решение подробнее (концентрируя внимание в основном на процессах типа pa), хотя в полной мере оно нас удовлетворить заведомо не может – ведь процесс типа pa должен «знать» типы потенциальных партнеров и разбираться с этими типами самостоятельно. Заодно познакомимся с вариантными типами Ады (в первом приближении вполне аналогичными паскалевским).

Решение с вариантным типом

```
package координаты_1 is
   n : constant integer :=3;
   type вид_процесса is(spa, spy, sp0);
   type pr;      -- предобъявление
   type ppr is access pr;
   task type pa is
      entry настройка(a : in real; слева, снизу : in ppr);
      entry сверху(X : in real);
      entry справа(Y : in real);
```

```
   end pa;
   task type px is
      entry настройка(j : in integer; снизу : in pa);
   end px;
   task type pb is
      entry настройка(bi : in integer; слева : in pa);
   end px;
   task type p0 is ...;
   task type py is ...;

   type pr(p : вид_процесса)    -- p — дискриминант
      record -- вариантного комбинированного типа pr
         case p is
            when spa=> sa:pa;      -- все поля различны!
            when spy=> sy:py;
            when sp0=> s0:p0;
         end case;
      end record;
end координаты_1;
package body координаты_1 is
   task body pa is
      aij, xj, yi, aij_xj : real;
      прт-слева, прт-снизу : ppr;  -- партнеры
   begin
      accept настройка(a : in real; слева.снизу : in ppr) do
         aij := a; прт-слева := слева;   прт-снизу := снизу;
      end настройка;
      ...
      accept сверху ...;
      ...
      accept справа ...;
      ...
      case птр-слева.p  is         -- proc_слева(птр-слева, yi);
         when spa => прт-слева.sa.справа(yi);
         when spy => прт-слева.sy.справа(yi);
         when sp0 => PUT("У p0 входа 'справа' нет.");
      end case;
         ...
      case прт-снизу.p is          -- proc_снизу(птр-снизу, xj);
         when spa => птр-снизу.sa.сверху(xj);
         when spy => PUT("У py входа 'сверху' нет.");
         when sp0 => птр-снизу.s0.сверху(xj);
      end case;
         ...
   end pa;
...   -- аналогично (но проще) для px, py, pb, p0
   для_pa : array(1..n, 1..n) of ppr; -- только
   для_py : array(1..n) of ppr; -- указатели
   для_p0 : array(1..n) of ppr; -- порождены
```

```
   для_px : array(1..n) of px; -- порождение и запуск процессов;
   для_pb : array(1..n) of pb; -- ждут настройки на pa
begin -- часть пакета, исполняемая при его загрузке
   -- порождение процессов
   for i in 1..n loop
      for j in 1..n loop
         для_pa(i,j) := new pr(spa);
      end loop;                   -- создано n**2 процессов типа pa и ссылки
   end loop;                      -- на них — в переменной для_pa ждут настройки!
   for i in 1...n loop
      для_py(i):= new pr(spy);    -- настройка не нужна (почему?)
   end loop; — ждут рандеву с pa
   for i in 1...n loop
      для_p0(i) := new pr(sp0);   -- настройка не нужна (почему?)
   end loop; -- ждут рандеву с pa
-- настройка процессов
      for i in 1...n loop         -- настройка процессов px на верхние pa
         для_px.настройка(j, для_pa(1,j)); -- доступ к ним не нужен,
      end loop;                   -- эти процессы сами заказывают рандеву
      ... -- аналогично предыдущему настройка pb
-- настройка процессов pa
   for i in 1…n-1 loop            -- кроме последней строки
      for j in 2..n loop          -- кроме первого столбца
         для_pa(i,j).sa.настройка(a(i,j), для_pa(i, j-1), для_pa(i+1,j));
      end loop;                   -- коммутация с остальными pa
   end loop;
   for j in 2..n loop
      для_pa(n,j).sa.настройка(a(n,j), для_pa(n,j—l),
      для_p0(j));                 -- коммутация с p0
   end loop;
   for i in 1..n-l loop
      для_pa(i,1).sa.настройка(a(i,l),
         для_py(i),               -- коммутация с py
            для_pa(i,1));
   end loop;
для_pa(n,1).sa.настройка(a(n.l),
   для_py(n), для_p0(1));         -- коммутация левого нижнего
end координаты_1;
```

Очевидны нерегулярность и сложность «вариантного» решения. Становится особенно наглядным различие в уровнях адекватности выразительных средств сущности параллелизма, достигнутых в Оккаме и в Аде. Решая средствами Ады проблемы, ранее успешно решенные средствами Оккама, мы убедились в том, что Ада-решение не только приводит к лишним затратам во время исполнения программы, но и существенно сложнее для понимания.

Упражнение (повышенной трудности). Придумайте более изящные Ада-решения рассмотренных проблем.

Подсказка. Сравните свои достижения с решением, изложенным на стр. 379.

5.9. Перечень неформальных теорем о параллелизме в Аде и Оккаме

Завершим рассмотрение концепции параллелизма в современных ЯП перечнем неформальных теорем, «доказательство» которых фактически представлено в предыдущих разделах. Читателю рекомендуется доказать (или опровергнуть) эти утверждения самостоятельно. Представим сначала утверждения, критикующие концепцию параллелизма в Аде, а затем и концепцию параллелизма в Оккаме.

Критика Ады «со стороны Оккама»

1. **Нет естественной симметрии между последовательным и параллельным исполнениями.** Другими словами, нельзя придать программе, в которой требуется и последовательное, и параллельное комбинирование процессов, регулярную структуру – приходится о параллелизме заботиться особо.
2. **Нет естественных средств идентификации асинхронных процессов.** Оккам позволяет не изобретать имена для каждого асинхронного процесса. Однако Ада вынуждает это делать. Чтобы прочувствовать, сколь обременительно подобное требование, представьте себе, что нужно изобретать имя для каждого оператора в Паскале.
3. **Нет естественных (учитывающих структуру программы) выразительных средств для запуска и завершения процессов.** И об этом приходится заботиться особо.
4. **Нет средств статической коммутации коллектива процессов**, точная структура которого неизвестна при создании программы.
   В Аде статическая коммутация ограничена явным указанием имен входов задач.
5. **Нет адекватной абстракции вида рандеву от процессов-партнеров** (ср. каналы и протоколы Оккама).

Критика Оккама «со стороны Ады»

Основная претензия – бедность «обычных» выразительных средств. Например:

1. **Нет определяемых программистом типов, нет исключений.**
   Однако самое интересное – сравнить эти ЯП в той области, для которой предназначен Оккам.
2. **Нет средств динамической коммутации процессов.**
   Хотя это можно объяснить стремлением к эффективности исполнения (в частности, стремлением возложить распределение процессов по физическим процессорам на транслятор), такое ограничение не позволяет работать с коллективами процессов, структура которых становится известной лишь при работе программы. Например, нельзя построить несбалансированное дерево процессов, учитывающее свойства сортируемого потока.

   Вопрос. Можно ли это сделать в Аде? Если можно, то как?

   Подсказка. Мы совсем недавно занимались динамической коммутацией процессов средствами Ады.

Таким образом, в Оккаме – высокая степень комфорта при программировании статических коллективов разнородных процессов. Поэтому типизация обмена (протоколы) отделена от процессов и связана с каналами. Ориентация на реальную многопроцессорную аппаратуру с эффективным контролем структуры программы. Оккамовскую программу можно «спаять» и запустить. И она должна быть максимально надежной. Для этого и нужны протоколы с квазистатическим контролем.

5.10. Единая модель временных расчетов

Некоторые существенные проблемы управления асинхронными процессами остались вне нашего рассмотрения. В качестве примера проблемы, оставшейся фактически открытой и в Аде, и в Оккаме, назовем **проблему управления временем исполнения процессов**.

Суть проблемы – в том, что программист должен иметь возможность оценить время исполнения критичных фрагментов программы и рассчитывать на соблюдение определенных гарантий в этом отношении. Такие оценки и гарантии должны быть выражены в терминах, независимых от среды исполнения (точнее, среда должна заранее предупреждать о невозможности предоставить требуемые программой гарантии).

Ситуация вполне аналогична управлению численными расчетами, однако никаких аналогов, касающихся времени исполнения процессов (не только параллельных), то есть своего рода *«единой модели временных расчетов», ни в Аде, ни в Оккаме нет*, хотя без ее решения программирование систем реального времени в терминах, независимых от среды исполнения, невозможно. Другими словами, это критичная технологическая проблема для языка реального времени, претендующего на достаточно высокий уровень абстракции.

Итак, в очередной раз содержательно работают общие критерии качества абстракции-конкретизации, позволяя прогнозировать появление в перспективе языковых средств, в совокупности предоставляющих в распоряжение программиста *единую модель временных расчетов*.

Подчеркнем еще раз, что дело не столько в самих средствах, позволяющих узнавать или заказывать допустимые интервалы времени для исполнения процессов, сколько в гарантии соблюдения заказанных интервалов в любой реализации ЯП (или обоснованного отказа принять программу к исполнению).

Считаю приятным долгом отметить, что внимание автора к рассмотренной проблеме было привлечено М. Ж. Грасманисом, занимавшимся проблемами параллелизма в ЯП под руководством автора в аспирантуре факультета ВМиК МГУ. С другой стороны, приведенная формулировка проблемы, аналогия с моделью числовых расчетов и соответствующий прогноз принадлежат автору.

5.11. Моделирование каналов средствами Ады

С учетом критики решения на стр. 374 рассмотрим решение задачи о преобразовании координат, опирающееся на динамическое моделирование соответствующей Оккам-программы средствами Ады. Преимущество такого решения – ясность и адекватность сути параллелизма. Наиболее очевидный недостаток – появление дополнительных процессов по сравнению с решением на стр. 374.

Ключевая идея. Ввести задачный тип «канал» в качестве активного мастера-посредника между «рабочими» процессами нашей программы, с тем чтобы все рабочие процессы стали равноправными клиентами каналов.

Во-первых, отпадет надобность в заказах рандеву с процессами разных типов. Во-вторых, отпадет надобность различать рандеву сверху-справа (как активные) и слева-снизу (пассивные) – все рандеву рабочих процессов становятся активными. (Кроме настройки! **Почему?**).

Схема решения на Аде:

1. Создаются коллективы процессов типа «канал». Для связи с ними используются массивы ссылок соответствующего типа. Каждый канал служит для связывания ровно двух процессов (как в Оккаме).
2. Создаются в нужном количестве процессы типов pa, px, py, pb, p0 и посредством рандеву-настройки с основным запускающим процессом настраиваются на соответствующие каналы.

```
package координаты_2  is
   n : constant integer := 3;
   task type канал;
   type на_канал is access канал;
   task type канал is
      entry ввод(X : in real);
      entry вывод(X : out real);
   end канал;
   верх_низ : array(1..n+1, 1..n) of на_канал;
   право_лево : array(1..n, 1..n+1) of на_канал;
   task type pa is
      entry настройка(a : in real; сверху, справа, снизу,
                             слева : in на_канал);
   end pa;
   task type px is
      entry настройка(j : in integer; снизу : in на_канал);
   end px;
   ...-- аналогично для py, p0, pb
end координаты_2;
package body координаты_2 is
   task body pa is
```

```
      aij, xj, yi, aij_xj : real;
      наверх, направо, вниз, влево : на_канал;
   begin
      accept настройка(a : in real; сверху, справа, снизу,
                           слева : in на_канал) do
         aij := a; наверх := сверху; направо := справа;
         вниз := снизу; влево := слева;
         -- присваивать каналы нельзя (почему?)
         -- приходится работать со ссылками на каналы
      end настройка;
      ...
   наверх.ввод(xj);   -- верх ? xj
      ...
   направо.ввод(yi);  -- право ? yi
      ...
   вниз.вывод(xj);    -- низ ! xj
      ...
   влево.вывод(yi);   -- лево ! yi
      ...
   end pa;
   для_pa : array(1..n, 1..n) of pa;
— процессы pa объявлены статически! Почему так можно?
   ... аналогично для px, py, pb, p0
begin — начало активного тела пакета
   — все "рабочие" процессы запущены и ждут настройки, но каналов еще нет!
   for i in 1..n+1 loop              -- создание каналов верх-низ
      for j in 1..n loop
         верх_низ(i,j) := new канал;
      end loop;
   end loop;                         -- ждут рабочих рандеву
   ... аналогично для каналов право_лево
         !! НАСТРОЙКА !!
   for i in 1..n loop                -- настройка px
      для_px(j).настройка(j, верх_низ(1,j));
                                     -- рандеву с процессом-родителем
   end loop;                         -- пошли первые рандеву с каналами
   for i in 1..n loop                -- настройка pb
      для_pb.настройка(b(i), право_лево(i, n+1);
   end loop;
   for i in 1..n loop
      for j in 1..n loop
         для_pa.настройка(a(i,j),
            верх_низ(i,j), право_лево(i,j+1),
            верх_низ(i,j+1), право_лево(i,j);
      end loop;
   end loop;
      ... аналогично для py и p0
end координаты_2;
      ... все работает
```

Между прочим, показан пакет с инициализирующими операторами, расположенными между begin и end его тела. Эта последовательность операторов работает в процессе «обработки» пакета (при его предвыполнении, при подготовке к нормальному предоставлению объявленных в нем услуг).

Подводя итог, видим, что моделирование каналов существенно упрощает описание и коммутацию коллективов однородных процессов. Каналы несложно моделировать в Аде, вводя дополнительные процессы. При этом, в отличие от нашего первого решения, не понадобились ссылки на «рабочие» процессы.

Вопрос. Почему?

Подсказка. Потому что их никому не нужно передавать.

Вопрос. Нельзя ли обойтись без ссылок на каналы?

Подсказка. Можно, если заставить процессы типа ра «знать» о различных массивах каналов либо ввести единый массив каналов (одновременно запутывая программу!).

Упражнение. Напишите соответствующий вариант программы.
Обратите внимание, фактически удается передавать параметры-каналы задачного типа. Процедурного типа в Аде нет. Наглядно видно, почему – подстановка разных экземпляров процессов одного типа не мешает контролю типов. А разных подпрограмм – может помешать!

Упражнение. Найдите недостатки предложенного решения.

5.12. Отступление о задачных и подпрограммных (процедурных) типах

Кажется удивительным, что в Аде асинхронный процесс служит объектом задачного типа, а обычный последовательный процесс (подпрограмма, процедура, функция) не может быть объектом данных вообще (не считается объектом какого-либо типа, «не имеет типа»). Поэтому, в частности, асинхронный процесс может быть параметром процедуры (и функции), а процедура и функция – не могут. Хотя, например, в Паскале допустимы процедурные типы, а также переменные и параметры таких типов.

Потребность в параметрах задачных типов была видна на примере параллельного преобразования координат (объекты типов ра, рх, ру... нуждались в настройке с параметрами, имеющими тип доступа к задачным типам).

Подчеркнем, что если бы в Аде разрешалось ввести процедурный тип, то в настройке достаточно было бы указать в качестве параметра процедуру-рандеву нужного партнера, и наше первое решение (без каналов) резко упростилось бы.

5.12.1. Входовые типы – фрагмент авторской позиции

Представим себе идеальное решение взаимодействия партнеров по параллельному преобразованию координат (без каналов, в рамках асимметричного рандеву).

Ключевая идея. Наши трудности были связаны с различием типов потенциальных партнеров. Однако при этом все процедуры-рандеву были вполне аналогичны как по своим ролям, так и по профилю. А именно их роль (назначение) – принять передаваемое (справа или сверху) вещественное число, а профиль имеет вид, например, (X : real) или просто (real), если не указывать имя формального параметра.

Поэтому было бы идеально ввести «входовый» тип нужного рандеву, а в каждом мастере-партнере объявить вход этого типа. Тогда клиента нужно было бы настраивать не на мастеров (различных типов), а на конкретные их входы (которые все одного входового типа). И никаких вариантных записей и операторов выбора (в зависимости от типа партнера)!

Выразительные средства

```
package координаты_3 is   -- координаты_1 – без каналов
      ...                 -- координаты_2 – с каналами
   entry type принять is(X : real); -- ключевой момент!
-- (Д1) в Аде этого нет! Предлагается так объявлять входовый тип с именем "принять"
и одним параметром X типа real. При этом X рассматривается в качестве атрибута
объекта типа принять.
   ...
   task type pa is
   entry настройка(a: in real; слева, снизу: принять);
-- (Д2) параметры типа "принять".
   entry сверху, справа : принять;
-- (Д3) это две константы типа "принять" – входы процесса типа pa
   end pa;
         ... – аналогично типы py и p0
   task type py is
         entry справа: принять;       -- (Д4) константа типа "принять"
      end py;
      task type p0 is
         entry сверху: принять;       -- константа типа "принять"
      end p0;
         ...  -- аналогично типы px, pb; но с учетом "настройки" для других, внешних
              -- партнеров.
end координаты_3;
package body координаты is
         task body pa is
            aij, xj, yi,... : real;   -- по-старому
            n_слева, n_снизу : принять;
-- (Д5) переменные типа "принять"; в них запоминается результат настройки
   begin
      accept настройка (a: in real; слева, снизу: принять) do
         aij := a; n_слева := слева; n_снизу := снизу;
-- (S1) передается "дескриптор" конкретного входа
      end настройка;
   ...
      accept сверху: принять do ... сверху.X... end сверху; -- (S2) почти по-старому
```

```
      ...
      accept справа: принять do ... справа.X... end справа;
      ...
      n_слева(yi); -- (S3) к нужному входу партнера слева,
                   -- а именно ко входу "справа", если настроить правильно.
      ...
      n_снизу(xj);
   end pa;
   task body py is
   ...
   begin
      accept настройка(...) do ... end;   -- для связей с внешним миром,
                                          --  которые здесь не рассмотрены
      ...
      accept справа: принять do ... справа.X... end справа;
   end py;
   ... -- аналогично для p0
   task body px is
     ...
   n_снизу : принять;
   begin
      accept настройка(j: in integer; снизу: принять) do
      ... n_снизу := снизу;
   end настройка;
      ...
   n_снизу(x);
      ...
   end px;
   ... -- аналогично pb
   end координаты_3;
```

Создание, запуск и коммутацию всех процессов можно выполнить аналогично случаю «координаты_2». Но при этом для настройки, например, нормального процесса типа pa с партнерами также типа pa достаточно применить оператор

```
для_pa(i,j).настройка(a(i,j), для_pa(i,j-1).справа, для_pa(i+1, j).сверху);
```

при этом передаются константы типа «принять», а именно входы «справа» и «сверху» партнеров соответственно слева и снизу.

Создание, запуск и коммутацию можно, как и раньше, выполнять либо в теле пакета, либо в использующем контексте. Например:

```
with координаты_3; use координаты_3;
procedure преобразование_координат is
   для_pa : array(1..n, 1..n) of pa;
   для_py : array(1..n) of py;
   для_p0 : array(1..n) of p0;
-- создаются нужные константы типа "принять"
   для_pb : array(1..n) of pb;
   для_px : array(1..n) of px; -- запуск процессов
begin
```

```
   ... затем настройка
end преобразование_координат;
```

Как видим, решение вполне регулярное, без лишних процессов (как в случае координаты_2) и без переборов вариантов (как в случае координаты_1). Все это – за счет «входового» типа «принять».

Уже сказано, что это частный случай подпрограммного типа. Предложенные выразительные средства вполне «в духе Ады». Ниже постараемся обосновать следующий тезис: *подобные средства отсутствуют в Аде, в частности потому, что ее авторы не заметили или не придали должного значения роли подпрограмм именно как однородных входов-рандеву разнородных асинхронных процессов.*

Другими словами, наш анализ выразительных средств Оккама и Ады выявил возможную точку роста языка Ада. Тем самым выявлена еще одна, возможно, критичная языковая потребность (например, для программирования транспьютеров).

Интересно отметить, что потребность в подпрограммных типах (со статически определенными профилями), а также в модели временных расчетов отмечена и в разработанных в 1990 г. требованиях к обновленной Аде (Ada9X Project Report. Draft Ada9X Requirements Document. August 1990. ISO-IEC/JTC1/SC22/WG9 #084).

5.12.2. Обоснование входовых типов

Проследим отношение к проблеме подпрограммных типов, привлекая в качестве иллюстрации ее решение в ряде ЯП (от Алгола-60 до Модулы-2).

В самом Алголе-60 было понятие класса параметров. В частности, был подпрограммный класс (точнее, класс процедур и класс функций). При этом ничего похожего на **Д1** не было, в **Д2** вместо ссылки на **Д1** нужно было бы указать класс параметров «слева» и «снизу», и только. В нашем случае этот класс был бы procedure. Так как профиль не был фиксирован, то и статический контроль согласованности вызовов и объявлений процедур был либо невозможен, либо, во всяком случае, требовал анализа всей программы. Например (следует текст почти на Алголе-60):

```
procedure P(F2,F2); procedure F1,F2;
begin real a,b; bool k,l;
   F1(a,b); F2(k,l); — нельзя проверить в рамках P
end;
procedure P1(c,d); real c,d; ...
procedure P2(F,c,d); procedure F; bool c,d:
      ...
   P(P1,P2);   P(P2,P1):    — неверный вызов.
```

Еще пример:

```
procedure P(F); procedure F;
begin
   if B then F(a, b)
      else F(c)
```

```
end;
procedure F1(k,l); . . .;
procedure F2; (m); . . .; -- обе можно подставлять
                          -- контроль только динамический.
```

В оригинальном (авторском) Паскале ситуация аналогична. В Алголе-68, где принята структурная совместимость типов (там они называются «видами»), аналог **Д1** не обязателен, нужные профили (структуры видов) указываются в аналогах **Д2** и **Д3**. При этом процедуры с аналогичными профилями считаются совместимыми. Отметим, что распознать такую «аналогичность» может оказаться невозможным (из-за появления в профилях ссылок вновь на процедурные типы, в том числе и рекурсивных ссылок).

В более поздних версиях Паскаля (в частности, в проектируемом новом стандарте ИСО и в ЯП Модула-2, наследовавшем лучшие черты Паскаля) введен и аналог **Д1** в виде

```
type принять = procedure(real);
```

и аналог **Д2**, и аналог **Д5**. Так что можно писать аналогично **S1** с тем же смыслом.

Однако аналога **Д3** и **Д4** в этих языках нет. Самое интересное для нас – в том, что это вполне закономерно.

Обратите внимание, мы ввели только аналог процедурного типа – тип входов. Они отличаются содержательно тем, что могут существовать одновременно и независимо, нуждаясь в индивидуальной идентификации как компоненты разных экземпляров процессов. При этом входы разнородных процессов способны играть аналогичные роли, что видно уже в момент их программирования (как в рассмотренном примере координаты_3). Поэтому их роль можно и целесообразно назвать специальным именем («принять»), считая его именем типа таких входов.

Итак, в случае входовых типов имя типа возникает естественно и применяется в полном соответствии с концепцией уникальности типа в Аде (то есть концепцией чисто именной согласованности типов). Важно, что и формально при этом не требуется ничего лишнего. В задаче имеется спецификация входа, где и указывается при необходимости его тип. Тем самым доказана неформальная **лемма**: *явное объявление «входовых» подпрограммных типов точно вписывается в концепцию типов языка Ада*. Обратите внимание, *мы с вами разработали обоснованное предложение по модернизации Ады*.

Упражнение. Найдите возражения против этого предложения.

Подсказка. Полезно рассмотреть не только (и не столько) возражения, опирающиеся на статус Ады как международного стандарта (хотя и стандарты пересматриваются), но и возражения, исходящие из принципа концептуальной целостности ЯП, – ведь входовые типы должны быть не инородной, а естественной частью ЯП, в полной мере оправдывающей свое появление в нем. Сверхзадача – ощутить себя в роли авторов (весьма нетривиального) ЯП, обязанных принять решение (с нетривиальными последствиями).

Верна и обратная **лемма**: *концепция типов языка Ада требует от подпрограммных типов явного объявления* (*как самого типа, так и принадлежности каждой подпрограммы этому типу*).

Действительно, тип параметра (любого, в том числе и подпрограммного типа) в Аде можно задать только посредством имени типа этого параметра. И если этот тип объявлен, например, так:

```
procedure type P is(X : real);
```

а некоторый параметр (или переменная) специфицирован так:

```
V : P;
```

то в общем случае конкретная процедура Q со спецификацией

```
procedure Q(X : real);
```

несовместима с V, если не указано явно, что она имеет процедурный тип P. Скажем, так:

```
Q : constant P is < тело >;
```

Сравните с Модулой-2, где Q совместима с V в силу структурной эквивалентности подпрограммных типов (явного типа P и анонимного типа процедуры Q). Но в Аде концепция типа – чисто именная. Обратная лемма доказана.

Итак, хотя для обычных процедур явная привязка к определенному типу выглядит обременительной (нужно придумать название типа и указать с ним связь при объявлении процедуры), концепция подпрограммных типов в целом (с явно именуемыми процедурными типами) вполне укладывается в типовую идеологию Ады.

Даже идея типа как содержательной характеристики роли объекта (в нашем случае – процедуры) проходит полностью.

Например:

```
procedure type P is(x : real);
Q1 : constant P is < тело1 >;
Q2 : constant P is < тело2 >;
Q3 : constant P is < тело3 >;
procedure type P1 is new P;  -- новая содержательная роль
procedure type P2 is new P;  -- еще одна.
```

С другой стороны, если процедуру не предполагается передавать в качестве параметра, то можно оставить и старый способ ее объявления, которое трактовать как объявление объекта анонимного подпрограммного типа.

Итак, с одной стороны, показаны естественность и полезность входовых типов, а с другой – показана возможность введения их как частного случая подпрограммных типов.

5.12.3. Родовые подпрограммные параметры

В Аде имеются родовые параметры (параметры периода компиляции), и подпрограммы могут выступать в их роли. Необходимость в таких параметрах диктуется следующими соображениями:

1. **Тип в Аде не может быть динамическим параметром** – это очевидным образом противоречит концепции статического контроля типа. Например:

```
procedure P(T1,T2) is  -- пусть T1 и T2 — параметры-типы
   a : T1;
   b : T2;
   a := b;             -- допустимо ли? Зависит от параметров.
```

2. **Потребность в настройке есть.** Например:

```
package стек is
   втолкнуть(X : in T);
   вытолкнуть(X : out T);
end стек;
```

Нужно сделать тип параметром! Однако из-за (1) – только статическим!

3. **Но тип в Аде – это класс значений вместе с классом операций.** Поэтому операции (подпрограммы) тоже должны быть параметрами (хотя бы статическими). Очередная неформальная теорема о необходимости в Аде родовых подпрограммных параметров доказана.

Хотелось бы сделать это экономно и надежно. Для этого следует:
1) обеспечить такой же уровень локального статического контроля модулей, как и без родовых параметров;
2) обеспечить возможность не перетранслировать родовые модули при каждой настройке на фактические родовые параметры.

Уже требование (1) приводит фактически к эквиваленту структурного подпрограммного типа, действующего в рамках родового модуля, – его роль в Аде играет спецификация родового процедурного параметра (см. примеры в разделе «Родовые сегменты»).

Упражнение. Обоснуйте последнее утверждение.

5.12.4. Почему же в Аде нет подпрограммных типов?

Основной наш тезис: *их нет потому, что ее авторы посчитали соответствующие технологические потребности удовлетворенными за счет родовых подпрограммных параметров* (*которые, как только что показано, в Аде все равно нужны*), *а также задачных типов.*

5.12.5. Заключительные замечания

Подведем итог, частично другими словами повторив сказанное выше.

1. **Однородные процессы, играющие аналогичные роли, естественны.** Контроль типов полностью статический, хотя объектов может быть неограниченно много.
2. **Однородные подпрограммы (с одинаковыми профилями) при отсутствии возможности динамически создавать подпрограммы остаются в ограниченном количестве и могут настраиваться статически.** Почти всегда достаточен аппарат родовых пакетов (хотя мог бы быть полезен аппарат репликаторов по примеру Оккама).

3. **Динамическое создание содержательно различных подпрограмм не соответствует идее концентрации контроля на богатой инструментальной машине.** Если пойти по этому пути, придется мощную систему контроля иметь на бедной целевой машине – а это не согласуется с требованиями к Аде.
4. **Коллективы асинхронных процессов полезны именно потому, что их члены взаимодействуют, работая «одновременно». Тип таких процессов естественно называть одним именем как в силу их аналогичного назначения, так и идентичного устройства.** Важно также, что идентификация объектов задачного типа – необходимое условие коллективного взаимодействия. При этом она не может быть непосредственно связана со статической структурой программы (примеры см. выше).
5. **Взаимодействие вызовов подпрограмм возможно только по исходным данным и результатам, так как исполнение одного вызова исключает исполнение другого** (случай рекурсивных вызовов – спорный). Передача доступа к такому объекту в качестве (динамического) параметра невозможна.
6. **Для обычных процедур привязка к определенному типу не только обременительна (нужно придумать название типа и явно указать с ним связь при объявлении процедуры), но в привычном адовском виде плохо согласуется с адовской же идеей типа** как характеристики содержательной роли значений этого типа.

   Действительно, если ввести P1 как имя типа для процедурного параметра V1, а имя P2 для типа параметра V2, то есть

   ```
   V1 : P1; V2 : P2;
   ```

   и по каким-то причинам хотеть, чтобы из набора конкретных процедур Q1, Q2, и Q3 вместо V1 можно было подставлять Q1 и Q2, а вместо V2 – Q2 и Q3, то пришлось бы дублировать тело Q2 – для объявления и с типом P1, и с типом P2.
7. Вот **пример воплощения средствами Ады параметрической интегрирующей функции**:

   ```
   generic
   with function f(X : in real) return real;
   function интеграл(A, B : in real) return real is
   ...
   end интеграл;
   function инт_f1 is new интеграл(f1);
   ```

8. **Глубинное различие задачи и подпрограммы** – в том, что если асинхронный процесс естественно понимать как экземпляр (объект) некоторого (задачного!) типа аналогичных объектов, то подпрограмма – это целый класс (тип) отдельных запусков. Именно отдельные запуски естественно считать экземплярами (объектами) подпрограммного типа.

   Итак, подпрограммный тип – это *тип типов вызовов*, задачный тип – *тип вызовов*. Другими словами, асинхронный процесс и подпрограмма различаются уровнем абстракции. Поэтому настраиваться на процесс существенно

проще, чем на подпрограмму. А именно следует учитывать, что у фиксированной подпрограммы остаются явные (динамические) параметры, что сама она фактически является типом (запусков), в отличие от асинхронного процесса.

Следовательно, в общем случае допуск процедур-параметров по сути означает допуск типов-параметров. Однако это уже очевидным образом противоречит статической концепции строгой типизации (**почему?**), принятой в Аде (и Паскале).

9. Но из этого затруднения авторы Ады и Паскаля выходят по-разному.

   Сказывается, *во-первых*, ориентация на чисто именную совместимость типов в Аде и на частично структурную – в Паскале. Действительно, в случае, когда профиль процедур фиксирован, вызовы различных процедур одного профиля укладываются в схему статического контроля типов (ведь именно профиль следует сопоставлять с типами фактических параметров и вырабатываемого результата параметрической процедуры или функции).

   Поэтому для Паскаля с его структурной концепцией типа естественно принять, что подпрограммы (процедуры и функции в терминах Паскаля) с одним профилем принадлежат одному типу (что не требует обязательных дополнительных указаний со стороны программиста).

   В Аде подобное допущение было бы неестественным, так как нарушило бы чисто именную концепцию типа.

   Во-вторых, следует учитывать отсутствие в Паскале иных возможностей настройки на процедурные параметры (совсем без нее не обойтись, достаточно вспомнить о процедуре интегрирования, зависящей от интегрируемой функции-параметра). А в Аде такая возможность есть – и процедуры, и типы могут служить родовыми параметрами. Другими словами, необходимая настройка может быть выполнена при компиляции.

Глава 6

Наследуемость (к идеалу развития и защиты в ЯП)

6.1. Определяющая потребность 392
6.2. Идеал развиваемости 392
6.3. Критичность развиваемости 393
6.4. Аспекты развиваемости 393
6.5. Идеал наследуемости (основные требования) 395
6.6. Проблема дополнительных атрибутов 395
6.7. Развитая наследуемость 398
6.8. Аспект данных 398
6.9. Аспект операций 401
6.10. Концепция наследования в ЯП (краткий обзор) 407
6.11. Преимущества развитой наследуемости 409
6.12. Наследуемость и гомоморфизм (фрагмент математической позиции) 410

6.1. Определяющая потребность

До сих пор мы интересовались в основном созданием программ «с нуля», почти не учитывая потребность использовать ранее предоставленные программные услуги для создания новых. Обеспеченность потребности «развивать» программные услуги (в ЯП, в программной среде, в средствах программирования в целом) будем называть «развиваемостью».

Эта потребность уже оказала серьезное влияние на современное программирование. Имеются основания считать, что в ближайшей перспективе это влияние станет определяющим. Во всяком случае, на развиваемость «работают» уже рассмотренная нами модульность, а также стандартизация, наследование, объектная ориентация, которые еще предстоит рассмотреть.

Другими словами, мы приступаем к обсуждению одной из фундаментальных концепций программирования.

6.2. Идеал развиваемости

Как известно, работающая, а тем более прекрасно работающая программа – это материализованный интеллект и немалый труд высококвалифицированных людей, который дорого стоит и к тому же содержит в себе элемент творчества (изобретения, открытия), результат которого в общем случае не может быть гарантированно воспроизведен в заданных условиях и в заданные сроки при любых мыслимых затратах. Поэтому развиваемость сознательно или интуитивно на протяжении всей истории информатики оставалась «голубой мечтой» создателей ЯП и иных средств программирования.

Так, еще на заре программирования появились библиотеки стандартных программ и средства модуляризации, отразившие максимальный для того времени уровень развиваемости. Однако в силу ряда причин, а скорее всего «по бедности и бескультурью», доминировало стремление предоставить средства заимствования, а не защиты от заимствования, и тем более не защиты заимствованного от разрушения. Примером такого относительно примитивного средства может служить оператор копирования «include».

На протяжении всей книги мы говорим о таком развитии, которое предполагает определенные гарантии защиты созданного от разрушения. Не зря в нашей концептуальной схеме в качестве взаимно дополнительных выделены именно средства развития и защиты абстракций, которыми мы и интересовались во всех наших моделях ЯП. Конечно, понятие развиваемости должно включать гарантию определенной защиты развиваемых программных услуг.

Это прежде всего защита авторского права. Она включает гарантированное предоставление специфицированных автором услуг в специфицированной автором среде и тем самым запрет на предоставление некоторых услуг, несанкционированных автором. Это, конечно, и защита потребителя, уже воспользовавшегося рассматриваемыми услугами. Его программы должны работать без каких-либо

дополнительных усилий и затрат. С другой стороны, естественно стремиться *к минимуму затрат на такое развитие программной услуги, которое не противоречит авторскому праву и правам потребителя*, – так мы сформулируем **идеал развиваемости**.

Примерами приближений к указанному идеалу могут служить классы и объекты в ЯП Симула-67, подробно рассмотренные нами пакеты в Аде, модули в Модуле-2. Как будет показано, все эти понятия не дотягивают до идеала. Вместе с тем они послужили экспериментальным основанием для современного воплощения идеала развиваемости.

6.3. Критичность развиваемости

Если со слабой развиваемостью еще можно было мириться в период первоначального накопления фонда программных услуг, то в настоящее время потребность в близкой к идеалу развиваемости приобретает характер критичной потребности для любого современного ЯП, поскольку фонд работающих, получивших признание программных услуг, уже создан практически во всех областях человеческой деятельности.

Более того, трудно представить себе такую область, где было бы нерационально воспользоваться программными услугами, хорошо проявившими себя в других областях, если при этом удается опереться на достаточно мощный аппарат развиваемости.

6.4. Аспекты развиваемости

Выделим три аспекта общего понятия развиваемости в ЯП (и современном программировании в целом): **модульность**, **стандартизация**, **наследуемость**.

Модульностью мы занимались в рамках модели А. Она обеспечивает развиваемость за счет фиксации сопряжения (интерфейса) между создателем и потребителем услуги. В результате создатель новой услуги может воспользоваться старой в рамках предписанного сопряжения (воспользовавшись «модулем» с правильно оформленным сопряжением). С другой стороны, старая услуга может быть заменена новой в рамках такого сопряжения, и пользователь получает новый комплекс услуг при определенной гарантии работоспособности комплекса в целом.

Итак, модульность способствует развиваемости «по частям», а тем самым повышает избирательность заимствования и снижает его стоимость. Модули обычно защищены от разрушения и несанкционированного использования предоставляемых ими услуг. Вместе с тем традиционные рамки модульности оказываются слишком жесткими, когда желательно заимствовать не весь модуль целиком, а с предварительной корректировкой некоторых услуг.

Для тех, кто привык пользоваться текстовыми редакторами, напомним, что желательно корректировать так, чтобы не «испортить» модуль, а этого простые текстовые редакторы не гарантируют.

Примеры модулей неоднократно приводились – это и спецификация модуля-пакета как сопряжение его реализации с его использованием, и спецификация модуля-процедуры, и т. п. Существенно, что до сих пор мы знали лишь такие модули, которыми можно пользоваться для обслуживания объектов, заранее неизвестных создателю модуля, но *тип таких объектов всегда известен* при трансляции (чаще при написании) модуля.

Вопрос. Для каких модулей тип обрабатываемых объектов неизвестен при трансляции?

Обратите внимание, что если статической типизации нет вообще, то трудно говорить о каких-либо гарантиях корректности, защите авторского права и т. п. Заметим, что статическая типизация может допускать и квазидинамический контроль типов (как в Симуле для контроля квалификаций ссылок, а в Аде – для контроля подтипов).

Стандартизация также предполагает фиксацию сопряжения между создателем и пользователем услуги, в частности сопряжения модулей с контекстом, однако основная цель при этом – не сам факт определения такого сопряжения, а *устранение нерационального разнообразия сопряжений*. Характерный пример – стандартизация ЯП: устраняется нерациональное разнообразие свойств различных реализаций одного и того же ЯП (обычно в различных программных средах). Таким образом, стандартизация принципиально расширяет «рынок» готовых услуг в пространстве и во времени, тем самым дополнительно стимулируя их производство и снижая удельные затраты на использование, – налицо обеспечение нового уровня развиваемости. Заинтересованного читателя отсылаем к [30–32].

Наследуемость – предмет настоящего раздела. Ее отличие от стандартизации ЯП – в том, что она не выходит за рамки одной программной среды. От традиционной модульности она отличается тем, что подразумевает существенно более гибкий аппарат заимствования, развития и защиты, действующий на уровне практически произвольных языковых объектов, а не только на уровне заранее предусмотренных модулей.

Такой уровень гибкости позволяет, в частности, легко приспособить программу к обслуживанию объектов, тип которых неизвестен не только при ее создании, но и при трансляции (с гарантией статического контроля типов).

Как мы увидим, это исключительно интересная концепция, одна из «изюминок» современного программирования, значительное приближение к идеалу развиваемости. Важно понимать, что этот аспект развиваемости ориентирован прежде всего на программистов, то есть создателей новых программных услуг (в отличие от стандартизации и частично модульности, которые ориентированы и на конечных пользователей непосредственно).

Примером продвижения в обозначенном направлении может служить наследуемость операций производных типов в Аде. Однако, как будет показано, она имеет существенные недостатки. Поразительна наследуемость в Симуле-67, в которой еще двадцать лет назад оказалось почти в точности то, что лишь сейчас осознается как *самое главное в программировании* (естественно, после принципи-

альной возможности программировать). Ближе к современному идеалу наследуемость в Обероне и Турбо Паскале 5.5 (а также в Смолтоке-5 и Си++). По сравнению с Симулой-67, прежде всего за счет эффективности и защиты.

Итак, **модульность** обеспечивает *упаковку* программных услуг в модули-контейнеры, **стандартизация** – *доставку* упакованных услуг потребителю-программисту в работоспособном состоянии, а **наследуемость** – *изготовление контейнера новых услуг с минимальными затратами, минимальным риском и в рамках законности*.

Конечно, каждый из трех названных аспектов опирается на тот или иной вариант абстракции-конкретизации.

Подчеркнем, что применяемая терминология – не общепринятая. Однако нам представляется, что, с одной стороны, наши термины достаточно точно отражают суть дела (с учетом смысла применяемых слов в русском языке), а с другой – в литературе на русском языке за обсуждаемыми понятиями термины еще не закреплены.

6.5. Идеал наследуемости (основные требования)

Содержательно **идеал наследуемости** звучит так: *программировать только принципиально новое*. Уточним его с учетом типизации языковых объектов. Должно быть возможно:

- определять новый тип, наследующий те и только те атрибуты исходного типа, которые желательны;
- пополнять перечень атрибутов нового типа по сравнению с перечнем атрибутов объектов исходного типа;
- гарантировать применимость сохраняемых операций исходного типа к объектам нового типа.

6.6. Проблема дополнительных атрибутов

Вернемся к принципу защиты авторского права, рассмотренному в связи с раздельной компиляцией. Суть его в том, что ЯП должен предоставлять программисту возможность *поставлять программный продукт, пригодный для санкционированного использования, но защищенный от использования несанкционированного* (в частности, от переделки программ или такого их «развития», при котором возникает опасность, что ранее работавшие программы работать перестанут). В Аде этот принцип поддержан отделением спецификации от реализации и возможностью не поставлять покупателю исходные тексты реализаций (тел пакетов, подпрограмм и задач).

Однако абсолютно запретить развитие программ неразумно. В Аде, в частности, имеются богатые средства развития (подпрограммы и определяемые типы). К сожалению, нетрудно привести примеры, когда эти средства оказываются недостаточными, если нежелательно или невозможно менять тексты реализаций. Подчеркнем, что такая ситуация возникает не только из-за недоступности текстов реализаций, но и из-за риска внести ошибку при их переделке.

Постановка задачи. Попытаемся, например, развить услуги пакета «управление_сетями» для сетей, обогащенных дополнительными атрибутами. Пусть для определенности каждый узел обладает некоторым «весом», содержательно соответствующим числу жителей в городе, представленном этим узлом.

Абстракция от реализации. Конечно, всегда можно воспользоваться методом, который мы применили, переходя от одной сети ко многим, – скорректировать сам пакет таким образом, чтобы удовлетворить новым потребностям.

Вопрос. Каковы недостатки предлагаемого пути?

Подсказка. Где гарантия, что старые программы – клиенты нашего пакета – останутся работоспособными после его модификации? Кстати, кто разрешил переделывать пакет? Ведь в приличном обществе его охраняет авторское право (к тому же исходные тексты тел могут оказаться недоступными – в соответствии с условиями контракта).

Вопрос. Каковы преимущества предлагаемого пути?

Подсказка. Не нужно осваивать новые средства программирования!

Итак, будем считать, что путь переделки комплекса программных услуг в связи с потребностью в новых услугах, казавшийся до сих пор привычным и самым естественным, – путь «революции», чреватый поломкой и того, что раньше неплохо работало, – путь неприемлемый. Постараемся подняться на новый уровень требований к средствам программирования, а именно потребуем «абстракции от реализации», чтобы можно было *развивать услуги, совершенно ничего не зная об их реализации*! Причем развивать по меньшей мере так, как нам потребовалось, чтобы можно было ввести сети с весомыми узлами.

Аспект данных проблемы атрибутов. Может показаться, что нечто подобное «мы уже проходили», когда шла речь о производных типах и наследовании в Аде. С этой точки зрения было бы естественным ввести «производный» тип сети_с_весом, наследующий все операции для типа «сети» из пакета управление_сетями и к тому же обладающий дополнительным атрибутом «вес_узлов».

Но в том-то и дело, что в Аде *невозможно естественно представить такой дополнительный атрибут*!

Вопрос. Почему?

Подсказка. Производные типы Ады по построению богаче родительских только за счет операций, а не за счет развития структуры данных.

Следовательно, атрибут «вес_узлов» придется вводить только за счет дополнительных операций, работающих с новым атрибутом. Это было бы вполне при-

емлемым, если бы для работы таких операций в структуре сети было заранее предусмотрено подходящее поле для хранения значения веса узла. Но ни о чем подобном мы не думали (и не могли думать), создавая тип «сети». Более того, даже если бы, предвидя потребность в развитии, мы предусмотрели «запасное» поле в таком типе, то строгая типизация заставила бы нас определить сразу же и тип его поля. Так что построить тип «сети_с_весом», не меняя пакета «управление_сетями», в Аде невозможно.

> **Вопрос.** Не помогут ли ссылочные типы? Ответ очевиден. Предоставляем его читателю.
>
> **Вопрос.** Не помогут ли родовые параметры?
>
> **Подсказка.** Формально они спасают положение – можно ввести запасной родовой параметр-тип специально для наращивания типа «сети» при развитии нашего пакета.

Читателю нетрудно оценить уровень изящества такого решения с учетом того, что запасные параметры придется вводить для любых типов в пакетах, претендующих на развитие рассматриваемого характера, пользоваться пакетом станет возможным только после конкретизации (возможно, с типами-заглушками вместо запасных типов, оказавшихся пока не развитыми), причем желательно требовать, чтобы транслятор не расходовал память для заглушек.

Конечно, всегда остается упомянутая выше (и отвергнутая) возможность написать новый пакет «управление_сетями_с_весом», воспользовавшись исходным пакетом как образцом. Но тогда уж производные типы (и вообще средства развития) окажутся ни при чем. Кстати, родовые сегменты частично автоматизируют именно такое переписывание, однако применимы и тогда, когда их исходные тексты недоступны. **Можно, конечно, ввести новый тип с полем «вес», не используя явно тип «сети». Но тогда для него придется определять и все операции, которые могли бы наследоваться. Другими словами, такое решение содержательно почти эквивалентно созданию нового пакета.**

Назовем описанную проблему развития услуг **проблемой дополнительных атрибутов**. Подчеркнем, что речь идет об атрибутах конкретных значений (экземпляров) типа, а не атрибутах типа в целом (к которым относятся, в частности, его базовые операции). Другими словами, конкретным весом обладает конкретный узел в конкретной сети, а не весь тип «сети». А вот, например, операция «связать» – атрибут всего типа «сети» – применима к любой паре узлов в произвольной сети.

Аспект операций проблемы атрибутов. Обратим внимание еще на один аспект проблемы дополнительных атрибутов. Мы пытались применить для ее решения средства наследования – производные типы Ады – именно потому, что желательно, чтобы старые операции над сетями были применимы и к обогащенным сетям, то есть сетям с весом. Другими словами, старые операции должны работать со значениями, структура которых неизвестна не только при создании определений этих операций, но и при их трансляции.

Вопрос. При чем здесь трансляция?

Подсказка. Мы должны учитывать возможное отсутствие даже исходных текстов пакета управление_сетями.

Более того, должны работать и все программы пользователя, написанные и оттранслированные до момента, когда он задумал работать с обогащенными сетями (с весом). Конечно, в этих программах не могут непосредственно использоваться операции с новыми именами – о них просто ничего не было известно при создании программ. Вместе с тем некоторые используемые в старых программах операции могут существенно зависеть от структуры и иных характеристик конкретных значений новых типов (например, для операции, показывающей сеть на экране, небезразличны количество и типы ее атрибутов).

Поэтому в общем случае решение проблемы дополнительных атрибутов должно предусматривать возможность подставлять в старые программы вместо старых тел операций их новые тела, учитывающие характеристики значений новых типов.

Указанный аспект проблемы назовем **аспектом операций** (в отличие от ранее рассмотренного аспекта данных). Подчеркнем, что аспект операций не сводится к аспекту данных, и наоборот – приходится предусматривать специфические средства развития операций.

Мы пришли к очередной неформальной **теореме**: *в Аде нет средств для адекватного решения проблемы дополнительных атрибутов*.

6.7. Развитая наследуемость

Итак, чтобы решить проблему дополнительных атрибутов (и тем самым воплотить близкую к идеалу гармонию между защитой работоспособности старых программ и оптимизацией усилий по их развитию), само понятие наследуемости необходимо развить по сравнению с наследуемостью в Аде.

Ключевые моменты такого развития: для аспекта данных – **обогащение типа** (возможность вводить дополнительные поля при объявлении производного типа), для аспекта операций – **виртуальные операции** (возможность вводить операции, заменяющие старые операции при действиях с обогащенными значениями, даже в старых программах). Последнее может показаться фантастичным, и тем не менее это всего лишь (частично) динамическая вариация на тему перекрытия операций.

6.8. Аспект данных

Покажем решение задачи обогащения сетей средствами Оберона (то есть минимальными средствами). Можно надеяться, что после этого будут легче восприниматься средства более мощных ЯП.

Чтобы избежать слишком подробных объяснений, сначала представим решение, потом прокомментируем.

```
DEFINITION УправлениеСетямиСВесом;
   IMPORT  У: УправлениеСетями, П: ПараметрыСети;
   CONST
      Удалить   = У.Удалить;
      Связать   = У.Связать;
      УзелЕсть = У.УзелЕсть;
      ВсеСвязи = У.ВсеСвязи;
(* объявление процедур-констант использовано для переименования *)
      (* Вопрос. Зачем оно нужно? *)
   TYPE
      Вес = SHORTINT;
      Узел = У.Узел;
      Сети = RECORD END;
   PROCEDURE Вставить(X: Узел; VAR ВСети : Сети; Р: Вес);
   PROCEDURE Присвоить(VAR Сеть1, Сеть2 : Сети);
   PROCEDURE ВесПути(X,Y: Узел; VAR ВСети : Сети): Вес;
   (* эти процедуры существенно зависят от обогащения*)
   END УправлениеСетямиСВесом;

MODULE УправлениеСетямиСВесом;
   IMPORT  У: УправлениеСетями, П: ПараметрыСети;
   CONST
      Удалить = У.Удалить;
      Связать  = У.Связать;
      Узел Есть = У.УзелЕсть;
      ВсеСвязи = У.ВсеСвязи;
   TYPE
      Вес = SHORTINT;
      Узел = У.Узел;
      Сети = RECORD(у.Сети) В:ARRAY П.МаксУзлов OF Вес
   END;
      (* обогащенные сети *)
      PROCEDURE Вставить(X : Узел; VAR ВСеть : Сети; Р: Вес);
      BEGIN
         У.Вставить(X, ВСеть); (*аргумент может быть обогащенным!*)
         ВСеть.В[X]:=Р;
      END Вставить;
      PROCEDURE Присвоить(VAR Сеть1, Сеть2 : Сети);
      BEGIN
         У.Присвоить(Сеть1, Сеть2);
         Сеть2.В := Сеть1.В;
      END Присвоить;
   (* Вопрос. Зачем оба параметра с режимом VAR? *)
   (* Подсказка. Стоит ли копировать сети? *)
   (* Вопрос. Зачем используются процедуры из пакета У? *)
   (* Подсказка. Без них — никак; вспомните о видимости. *)
   PROCEDURE ВесПути(X.Y: Узел; VAR ВСети : Сети): Вес;
   ...
   BEGIN
```

```
   ...
   RETURN Вес;
   END ВесПути;
END УправлениеСетямиСВесом;
```

Упражнение. Запрограммируйте функцию ВесПути, используя процедуру ВсеСвязи.

Итак, мы написали модуль, позволяющий работать с обогащенными сетями и при этом использующий (и сохраняющий) все средства работы со старыми сетями. Самое для нас существенное – ничего не потребовалось знать об их реализации; мы пользовались только видимыми пользователю атрибутами старых сетей! Заново пришлось программировать лишь операции, существенно использующие новые атрибуты, но и при этом было удобно пользоваться операциями над старыми сетями – надежно и естественно – без нарушения авторского права.

Конечно, можно еще более сократить дополнительные усилия и не вводить переименований старых имен. Однако пользоваться обогащенными сетями было бы менее удобно (за счет чего?). Кстати, переименования и не потребовались бы, если бы обогащенный тип «сети» мы ввели прямо в модуль УправлениеСетями (полезное упражнение). Однако хотелось показать, как задача полностью решается чисто модульными средствами Оберона без всякой перетрансляции исходного модуля.

Сделаем еще ряд технических пояснений, хотя можно надеяться, что пример уже полностью понятен. Обратите внимание, что в реализации модуля заново повторяется все, что указано в спецификации (в отличие от Ады и Модулы-2). Этого требует принцип экспортного окна. Существенно эксплуатируется возможность обращаться к старым процедурам с аргументами обогащенного типа **(где, например?)** – при этом старые процедуры используют и модифицируют только старые поля, хотя, с другой стороны, прямые присваивания старых объектов новым в Обероне запрещены!

Вопрос. Зачем такой запрет?

Подсказка. Вспомните о надежности.

Может показаться, что было бы естественнее обогащать не сети, а узлы или записи_об_узле. Однако в каждом из этих случаев возникают технические препятствия, заслоняющие суть дела и потому мешающие привести такое обогащение в качестве простого примера. Во-первых, тип «узел» не является комбинированным, а только для комбинированных типов в Обероне допустимо обогащение. Во-вторых, тип запись_об_узле скрыт в теле модуля, и поэтому нельзя его обогащать извне модуля.

Наконец, обогащение типа «узел» формально никак не скажется на построенных на его основе составных типах – они такого обогащения «не заметят» (это справедливо, конечно, и для типа «запись_об_узле»).

Вопрос. Как сделать, чтобы «заметили»?

Подсказка. Посмотрите на раздел констант модуля УправлениеСетямиСВесом.

Вопрос. Можно ли при этом обойтись без перетрансляции модуля Управление_ Сетями?

Подсказка. Ответ очевиден. И даже если предварительно выделить модуль, определяющий, например, тип «запись_об_узле» (пренебрегая защитой!), то придется вспомнить о порядке раздельной трансляции.

Замечание. Фактически мы проявили «точку роста» наследуемости, пока, насколько известно, не использованную авторами ЯП. Аналогично тому, как старые операции воспринимают обогащенные структуры, так и «старые» структуры могли бы воспринимать свои обогащенные компоненты, не требуя переименований и перетрансляций. Точнее говоря, такие «транзитивные обогащения» должны быть допустимыми и без перетрансляции, но последняя, возможно, окажется полезной для оптимизации расхода ресурсов.

И вообще полезно понимать, что ожидаемые *преимущества от нового стиля программирования даются не бесплатно*. Мы и раньше обращали внимание на тот факт, что решение критичных проблем требует как соответствующих выразительных средств, так и методики их применения для решения выделенной проблемы. В частности, чтобы иметь возможность развивать программные услуги с учетом идеала наследуемости, нужно заранее позаботиться о строении предназначенных для такого развития услуг. В этом и проявляется определенная методика программирования.

Упражнение (повышенной трудности). Перепишите модуль УправлениеСетями так, чтобы было возможно обогащать узлы или записи об узле, а затем напишите соответствующие обогащения.

6.9. Аспект операций

Мы рассмотрели такое обогащение сетей, при котором новый атрибут не влияет на осмысленность старых операций. Точнее говоря, мы были вынуждены заменить определения операций «вставить» и «присвоить», однако при этом с успехом пользовались в новом контексте и старыми одноименными операциями.

Но нетрудно указать на такие исходные типы и такие их обогащения, когда исходные операции совершенно теряют смысл. Классический пример – тип «окружность» с операциями вывода на экран или принтер, и обогащение «дуга окружности» с новыми атрибутами «начальный_угол» и «величина дуги». Ясно, что печать дуги на принтере нельзя (или трудно, неестественно) представить через печать полной окружности – это противоречит самой сути новых атрибутов (предназначенных как раз для вырезания лишь части окружности).

Назовем подобные обогащения **ограничивающими**. Ясно, что для ограничивающих обогащений необходимо иметь механизм полной замены исходных операций над объектами старых типов.

В Обероне специального аппарата для подобной цели нет. Однако можно воспользоваться переменными процедурного типа. Тем более что такие переменные могут быть и полями объектов комбинированного типа.

Например, исходный модуль УправлениеСетями может иметь вид:

```
DEFINITION УправлениеСетями;
   IMPORT П: ПараметрыСети;
   TYPE
      Узел = SHORTINT;
      ПереченьСвязей = ARRAY П.МаксСвязей OF Узел;
      Связи = RECORD
                  Число : SHORTINT;
                  Узлы : ПереченьСвязей;
              END;
      Сети = RECORD END;
   VAR (* переменные процедурных типов *)
      УзелЕсть: PROCEDURE(Узел, VAR Сети) : BOOLEAN;
      ВсеСвязи    : PROCEDURE(Узел, VAR Сети, VAR Связи);
      Вставить: PROCEDURE(Узел, VAR Сети);
      Присвоить : PROCEDURE(VAR Сети, VAR Сети);
      Связать  : PROCEDURE(Узел, Узел, VAR Сети);
      Удалить  : PROCEDURE(Узел, VAR Сети);
END УправлениеСетями;
MODULE УправлениеСетями;
   IMPORT П: ПараметрыСети;
   TYPE
      Узел = SHORTINT;
      ПереченьСвязей = ARRAY П.МаксСвязей OF Узел;
      Связи = RECORD
                  Число : SHORTINT;
                  Узлы : ПереченьСвязей;
              END;
      ЗаписьОбУзле = RECORD
                  Включен : BOOLEAN;
                  Связан :Связи;
                    END;
      Сети = RECORD С: ARRAY П.МаксУзлов OF ЗаписьОбУзле
END;
   VAR (* переменные процедурных типов *)
      УзелЕсть: PROCEDURE(Узел, VAR Сети) : BOOLEAN;
      ВсеСвязи    : PROCEDURE(Узел, VAR Сети, VAR Связи);
      Вставить: PROCEDURE(Узел, VAR Сети);
      Присвоить : PROCEDURE(VAR Сети, VAR Сети);
      Связать  : PROCEDURE(Узел, Узел, VAR Сети);
      Удалить : PROCEDURE(Узел, VAR Сети);
      (* ниже следуют константы соответствующих процедурных типов*)
   PROCEDURE УзелЕсть1(X: Узел; VAR ВСети : Сети) : BOOLEAN;
   BEGIN
      RETURN ВСети.С[X].Включен;
   END УзелЕсть1;
   PROCEDURE ВсеСвязи1(X : Узел; VAR ВСети : Сети; VAR R: Связи);
   BEGIN
      R := ВСети.С[X].Связан;
```

```
END ВсеСвязи1;
PROCEDURE Вставить1(X : Узел; VAR ВСеть : Сети);
BEGIN
   ВСеть.C[X].Включен:=TRUE;
   ВСеть.C[X].Связан.Число:= 0;
END Вставить1;
PROCEDURE Присвоить1(VAR Сеть1, Сеть2 : Сети);
BEGIN
   Сеть2.C:= Сеть1.C;
END Присвоить1;
PROCEDURE Есть_связь(АУзел, ВУзел : Узел, VAR ВСети : Сети): BOOLEAN;
   VAR i : 1..П.МаксСвязей;
         z: Связи;
BEGIN
   z := ВСети.C[АУзел].Связан;
   i:=0;
   REPEAT (* цикла FOR в Обероне нет *)
   IF z.Узлы(i)= ВУзел THEN
      RETURN TRUE;
   END;
   i := i + 1;
         UNTIL i < z.Число
         RETURN FALSE;
END Есть_связь;
PROCEDURE Установить_связь(Откуда, Куда : Узел; VAR ВСети : Сети);
   VAR z: Связи;
BEGIN
   z := ВСети.C[АУзел].Связан;
   z.Число:= z.Число+1;
   z.Узлы(z.Число) := Куда;
END Установить_связь;
PROCEDURE Связать1(АУзел, ВУзел : Узел; VAR ВСети : Сети);
BEGIN
   IF ~ Есть_связь(АУзел, ВУзел, ВСети) THEN
(* «~» — отрицание *)
      Установить_связь(АУзел, ВУзел, ВСети);
         IF АУзел # ВУзел THEN
   (* «#» в Обероне — знак неравенства *)
            Установить_связь(ВУзел, АУзел);
         END;
   END;
END Связать 1;
PROCEDURE Переписать(ВУзле : Узел; После : SHORTINT; VAR ВСети : Сети);
   VAR j : SHORTINT;
BEGIN
   j := После;
   WHILE J > ВСети.C[ВУзле].Связан.Число-l DO
   ВСети.C [ВУзле].Связан.Узлы[j]:= ВСети.C[ВУзле].Связан.Узлы [j+1];
      j:= j+i;
```

```
      END
   END Переписать;
   PROCEDURE Чистить(Связь, ВУзле : Узел; VAR ВСети : Сети);
      VAR i: SHORTINT;
   BEGIN
      i:= 0;
      REPEAT
         IF ВСети.С[ВУзле].Связан.Узлы[i]=Связь THEN
            Переписать(ВУзле, i, ВСети);
            ВСети.С[ВУзле].Связан.Число := ВСети.С[ВУзле].Связан.Число-1;
            EXIT;
         END; i := i+1;
      UNTIL i < ВСети.С[ВУзле].Связан.Число
   END Чистить;
   PROCEDURE Удалить1(X : Узел; VAR ИзСети : Сети);
      VAR i : SHORTINT;
   BEGIN
      ИзСети.С[X].Включен:=FALSE; i:=0;
      REPEAT (* цикла FOR в Обероне нет *)
         Чистить(X, ИзСети.С[X].Связан.Узлы[i], ИзСети);
         i := i+1;
      UNTIL i < ИзСети.С[X].Связан.Число
   END Удалить 1;
BEGIN (* Инициализация — работает при загрузке модуля *)
   УзелЕсть   := УзелЕсть1;
   ВсеСвязи   := ВсеСвязи 1;
   Вставить    := Вставить1;
   Присвоить := Присвоить 1;
   Связать :=Связать1;
   Удалить := Удалить1;
END УправлениеСетями;
```

А модуль УправлениеСетямиСВесом принимает вид

```
DEFINITION УправлениеСетямиСВесом;
   IMPORT У: УправлениеСетями, П: ПараметрыСети;
   TYPE
      Вес = SHORTINT;
      Узел = У.Узел;
      Сети = RECORD END;
   VAR
      УзелЕсть: PROCEDURE(Узел, VAR Сети) : BOOLEAN;
      ВсеСвязи :PROCEDURE(Узел, VAR Сети, VAR Связи);
      Вставить: PROCEDURE(Узел, VAR Сети, Вес);
      Присвоить : PROCEDURE(VAR Сети, VAR Сети);
      Связать : PROCEDURE(Узел, Узел, VAR Сети);
      Удалить : PROCEDURE(Узел, VAR Сети);
      ВесПути: PROCEDURE(Узел, Узел, VAR Сети): Вес;
END УправлениеСетямиСВесом;
```

```
MODULE УправлениеСетямиСВесом;
   IMPORT У: УправлениеСетями, П: ПараметрыСети;
   TYPE
      Вес = SHORTINT;
      Узел = У.Узел;
      Сети = RECORD(У.Сети) В: ARRAY П.МаксУзлов OF Вес
   END;;
      (* обогащенные сети *)
   VAR
      УзелЕсть: PROCEDURE(Узел, VAR Сети) : BOOLEAN;
      ВсеСвязи    : PROCEDURE(Узел, VAR Сети, VAR Связи);
      Вставить: PROCEDURE(Узел, VAR Сети, Вес);
      Присвоить : PROCEDURE(VAR Сети, VAR Сети);
      Связать  : PROCEDURE(Узел, Узел, VAR Сети);
      Переписать: PROCEDURE(Узел, SHORTINT, VAR  Сети);
      Удалить : PROCEDURE(Узел, VAR Сети);
      ВесПути : PROCEDURE(Узел, Узел, VAR Сети): Вес:

(* ниже следуют процедурные константы, заменяющие исходные *)
   PROCEDURE Вставить1(X : Узел; VAR ВСеть : Сети; P: Вес);
   BEGIN
      У.Вставить(X, ВСеть);
      (* аргумент может быть обогащенным! *)
      ВСеть.В[X] := P;
   END Вставить1;
   PROCEDURE Присвоить1(VAR Сеть1, Сеть2 : Сети);
   BEGIN
      У.Присвоить(Сеть1, Сеть2);
      Сеть2.В := Сеть1.В;
   END Присвоить1;
   PROCEDURE ВесПути 1(X,Y: Узел; VAR ВСети : Сети): Вес,
   ...
   BEGIN
   ...
   RETURN Вес;
   END ВесПути1;
BEGIN (* Инициализация — работает при загрузке модуля *)
   УзелЕсть := У.УзелЕсть;   (* старая *)
   ВсеСвязи := У.ВсеСвязи;   (* старая *)
   Вставить := Вставить 1;   (* новая *)
   Присвоить := Присвоить 1; (* новая *)
   Связать := У.Связать;     (* старая *)
   Удалить := У.Удалить;     (* старая *)
   ВесПути := ВесПути1;      (* новая *)
END УправлениеСетямиСВесом;
```

Таким образом, появляется возможность при обогащении устанавливать значения нужных процедур и в старом, и в новом модулях. Однако имеются два не-

удобства: во-первых, в старом модуле невозможно предусмотреть тип «процедур будущего». Например, в нашей новой процедуре «Вставить» появился новый параметр, и в результате типы старой и новой процедур несовместимы. Во-вторых, нет явной связи обогащенного типа именно с новыми операциями – можно по ошибке применить и старые операции, причем диагностики никакой не будет (ведь, возможно, этого и хотелось!).

В принципе, как уже сказано, Оберон позволяет связать нужные операции и с каждой сетью индивидуально. Для этого можно использовать поля процедурного типа в объекте типа «сети_с_процедурами», который можно объявить следующим образом:

```
TYPE
   Сети = RECORD(У.Сети)
      B: ARRAY П.МаксУзлов OF Вес
      УзелЕсть: PROCEDURE(Узел, VAR Сети) : BOOLEAN;
      ВсеСвязи : PROCEDURE(Узел, VAR Сети, VAR Связи);
      Вставить: PROCEDURE(Узел, VAR Сети, Вес);
      Присвоить : PROCEDURE(VAR Сети, VAR Сети);
      Связать : PROCEDURE(Узел, Узел, VAR Сети);
      Переписать: PROCEDURE(Узел, SHORTINT, VAR  Сети);
      Удалить : PROCEDURE(Узел, VAR Сети);
      ВесПути : PROCEDURE(Узел, Узел, VAR Сети): Вес;
END;
```

Тем самым в Обероне обеспечен и необходимый минимум средств для «настоящего» объектно-ориентированного программирования. Так что в «чемоданчике» и на этот случай многое предусмотрено. Конечно, аналогичные приемы применимы и к Модуле-2, но там придется моделировать и встроенный в Оберон механизм обогащения типов.

> **Упражнение.** Найдите способ программировать в объектно-ориентированном стиле средствами Модулы-2.
>
> **Подсказка.** Процедурные типы в Модуле-2 имеются.

Правда, отсутствие соответствующего управления видимостью не позволяет средствами Оберона легко и естественно выполнять привязку конкретных процедур к конкретным объектам. Например, процедура с именем «Вставить» в общем случае никак не связана с полем «Вставить» ни в объявлении типа «Сети», ни с полем «Вставить» в объекте, например, «Сеть1». Приходится воплощать необходимые связи явными присваиваниями (например, процедуры Вставить полю Сеть1.Вставить). Нужные средства (в том числе и управления видимостью) предоставляют более мощные ЯП с развитой наследуемостью. Среди них – Турбо Паскаль 5.5, примеры программ на котором мы рассмотрим в следующем разделе.

6.10. Концепция наследования в ЯП (краткий обзор)

Накоплено достаточно материала, чтобы поговорить об общих свойствах наследуемости в ЯП. Сделаем это в форме изложения элементов частично формализованной концепции (теории) наследования и краткого обзора воплощения этой концепции в современных ЯП.

6.10.1. Основные понятия и неформальные аксиомы наследования

Основные понятия:

- отношение наследования (короче – **наследование**) между родителем и наследником; например отношение между типами «Сети» и «СетиСВесом»;
- атрибуты наследования (**атрибуты**); например процедуры Удалить и др.;
- разность атрибутов наследника и родителя (**накопление**); например процедуры ВесПути и Вставить;
- **типы объектов** и **экземпляры** (объекты) определенных типов; например «Сети» и «Сеть1».

В этих терминах сформулируем аксиомы наследования, проявляя связи перечисленных понятий (и тем самым определяя последние). Главная цель пункта – уточнить понятия и подготовиться к изложению математической концепции (модели) наследования.

Отношение наследования определяется для типов, а не экземпляров (объектов). Обратите внимание, в живой природе – наоборот: индивиды-наследники наследуют свойства индивидов-родителей! Стоит задуматься над причиной такого различия, а может быть, и усмотреть интересные перспективы.

Наследник обладает всеми атрибутами родителя. Обратное неверно.

Право участвовать в операциях определенного класса – это атрибут наследования. *Следствие*: наследник имеет право замещать родителя в любых таких операциях.

Вопрос об определении указанного класса непрост. В Обероне присваивание в него попадает лишь частично – обогащенным объектом можно замещать источник присваивания, но не получатель.

Все экземпляры (объекты) одного типа обладают идентичными атрибутами (но не их значениями!).

Следствие: индивидуальные свойства объектов (определяемые, в частности, значениями атрибутов) не наследуются и по наследству не передаются.

Наличие наследников не влияет на атрибуты родителя.

Следствие: свойство «иметь наследников» не считается атрибутом (и, стало быть, не наследуется).

Атрибуты могут быть абстрактными (виртуальными) и тем самым требовать конкретизации (настройки на контекст перед использованием или в процессе использования), или конкретными, и тем самым такой настройки не требовать.

Пример абстрактного атрибута – поле «С» в типе «Сети». Его конкретизация – поле с именем «С» в конкретном объекте типа «Сети», например Сеть1.С. Пример конкретного атрибута типа «Сети» – процедура Удалить.

Результат конкретизации абстрактных атрибутов в общем случае определяется тем контекстом, где объекты создаются. *Следствие*: поскольку в этот контекст входят не только типы, но и конкретные объекты (экземпляры типов), то конкретные значения абстрактных атрибутов могут зависеть от свойств конкретных объектов контекста, а не только от их типов.

Настройку на контекст естественно выполнять при создании (инициализации) объектов. Например, объект-очередник при создании получает конкретное значение атрибута, указывающего на стоящего впереди объекта.

Накопление в некоторых ЯП может состоять как из абстрактных, так и из конкретных атрибутов (Смолток, Оберон, Турбо Паскаль 5.5), **а в других – только из конкретных атрибутов (Ада)**.

Следствие: в ЯП второй категории значения атрибутов не могут учитывать свойства (и даже сам факт существования) объектов таких типов, которые не существовали в момент написания (трансляции) программы.

Итак, операции-атрибуты Ады могут служить примерами конкретных атрибутов. Они принадлежат только производному типу, а не индивидуальным объектам этого типа в том смысле, что значения этих атрибутов (то есть конкретные функции и процедуры) никак не зависят от экземпляров объектов производного типа, не связаны с ними по происхождению и не устанавливаются при создании объектов. Более того, *из-за ограничений строгой типизации в Аде нельзя построить тип объектов, какие-либо производные которого могли бы содержать ссылки (указатели) на экземпляры объектов производного типа*.

> **Упражнение.** Докажите неформальную теорему, сформулированную в последнем предложении предыдущего абзаца.

Примерами абстрактных атрибутов могут служить, *во-первых*, атрибуты-поля комбинированных типов (в любых ЯП, где есть наследование). Их естественно считать абстрактными именно потому, что с типом связаны только имена полей, а конкретные значения этих атрибутов (то есть конкретные поля конкретных экземпляров объектов – **не путать со значениями, которые можно присваивать этим полям!**) возникают только при создании объектов. Другими словами, их связь с объектами индивидуальна.

В тех ЯП, где атрибуты-поля входят в накопление (Симула-67, Оберон, Турбо Паскаль 5.5 и др.), в объекты производных типов могут входить поля, содержащие (или ссылающиеся на) объекты производных типов. Именно потому, что эти поля формируются в контексте, где уже доступны обогащенные (производные) типы,

строгая типизация не мешает их формированию, в отличие от ситуации, когда в накоплении допустимы только конкретные атрибуты (Ада).

Во-вторых, примерами абстрактных атрибутов служат виртуальные операции (операции Симулы-67, правила (методы) Турбо Паскаля 5.5, Смолтока и др.). Подробнее о них – в следующем разделе.

Итак, введенные понятия позволяют проявить весьма существенные различия концепции наследования в современных ЯП. Например, в Модуле-2 наследование отсутствует совершенно (хотя можно усмотреть его зачатки в связи модулей по импорту), в Аде – наследование только с конкретным накоплением, в Обероне – возможно и абстрактное накопление, но нет виртуальных операций. Наконец, в Турбо Паскале 5.5 нас ждет почти идеальное наследование. «Почти» – потому, в частности, что имеются точки роста, – вспомните недавнее замечание об обогащении составных типов.

С другой стороны, вдумчивого читателя еще со времени первого знакомства с Обероном, возможно, мучит вопрос о правомерности (или неправомерности) присваивания бедных объектов обогащенным и вообще о применимости старых операций к новым объектам.

Можно, конечно, искать общий ответ, и мы предложим на этот счет некоторые соображения. Однако не менее интересно и поучительно убедиться, что обозначенная проблема исчезает совершенно, если в центр внимания поместить не операции, а объекты, что и сделано в объектно-ориентированном программировании. Действительно, пусть сами объекты «решают», с кем им нужно взаимодействовать, а с кем – нет.

Прежде чем подробнее заняться объектно-ориентированным программированием, укажем еще раз на преимущества развитой наследуемости.

6.11. Преимущества развитой наследуемости

Гармония открытости и защиты. Принцип защиты авторского права реализуется в естественной гармонии с принципом открытой системы. Последний состоит в том, что пользователь не только получает доступ к декларированным возможностям системы, но и может перестраивать систему для своих нужд способом, не предусмотренным явно ее создателем.

Гармония состоит в том, что, с одной стороны, развитый аппарат наследования позволяет создавать системы, пригодные в сущности для развития в совершенно произвольном направлении путем пошаговых преобразований, а с другой – ни одно из этих преобразований не способно нарушить корректное исполнение старых программ (и вообще предоставление услуг, декларированных создателем программы). Другими словами, *идеальная гибкость сочетается с идеальной надежностью* (при вполне приемлемой эффективности и близких к оптимальным трудозатратах за счет потенциального роста тиражируемости программных изделий).

Поддерживаемая развитой наследуемостью технология развития программной системы (пошаговая модификация работающей основы) ***способствует оптимизации и структуризации мышления, программирования, памяти***.

Эта технология полностью согласуется с концепцией расслоенного программирования А. Л. Фуксмана [33], описанной им более десяти лет назад (пошаговое наращивание новых «слоев» работающей версии программы от минимальной работоспособной основы до цели – разветвленной системы услуг).

Поддерживаемый стиль мышления адекватен естественному развитию от простого к сложному, от общего к частному, от единого корня к разнообразию. (В отличие от классического структурного программирования, подразумевающего лишь пошаговую детализацию сверху вниз.)

Говоря более конкретно, ***развитое наследование обеспечивает расширяемость объектов, типов и операций с защитой авторского права и с гарантией сохранности старых программ***. Можно воспользоваться программой, развивать ее, но нельзя украсть «секреты фирмы», если они скрыты автором.

В заключение подчеркнем *существенное отличие изложенного понятия наследования от аналогичного понятия из «реальной жизни»*. В последнем случае не только сами типы, но и конкретные экземпляры объектов таких родительских типов, как «животные», «кошки», «собаки» и т. п., реально существуют лишь как некоторые абстракции, представленные, например, совокупностью генов или изменчивым набором ныне живущих особей (обладающих, конечно, неисчислимым множеством свойств, никак не охватываемых соответствующим типом). А в информатике и типы, и экземпляры объектов представлены вполне реальными разрядами, записями, фреймами и т. п. Соответственно, и наследование пока представлено как определение и обработка определений типов, а не результат «жизнедеятельности» объектов. Однако нет оснований полагать, что так будет всегда.

6.12. Наследуемость и гомоморфизм (фрагмент математической позиции)

Преамбула. В разделе об авторской позиции было предложено упражнение (повышенной трудности) – догадаться, какое известнейшее математическое понятие непосредственно связано с наследуемостью. Конечно, речь идет о гомоморфизме. Не исключено, что для активно интересующихся проблемами программирования математиков такая связь давно очевидна, однако автору не приходилось встречать упоминание об этом в литературе.

Открытие связи наследования с гомоморфизмом принадлежит В. А. Левину. Когда он впервые рассказал на нашем семинаре в МГУ о языке спецификаций Атон [34], существенно уточняющем и развивающем V-подход [35], я обратил его внимание на отсутствие в Атоне аппарата наследования. Для меня особый интерес к наследованию как к одной из фундаментальных концепций программирования в этот период был естествен – как раз вызревал соответствующий раздел книги.

А для В. А. Левина, занятого математической формализацией содержательных концепций спецификации в Атоне, оказалось совершенно естественным понять, какое именно математическое понятие следует сопоставить наследованию. Буквально на следующий день он предложил мне уже упомянутое упражнение и, выдержав небольшую паузу, выдал результат, после чего мы стали наперебой обсуждать и другие возможные роли гомоморфизма в программировании. Так что следующие ниже соображения можно считать результатом наших совместных обсуждений.

Суть дела. Для начала напомним, что **гомоморфизм** g из области отправления P в область прибытия Q – это отображение

```
g: P-->Q
```

типа P -->Q, где P и Q – алгебры (алгебраические системы), для которого выполнено определяющее соотношение

```
g*f= g(f)*g
```

для всякой операции f из P (звездочкой обозначена композиция отображений). Для краткости и ясности принято, что все операции унарны. При необходимости n-ки аргументов n-арных операций нетрудно заменить составными объектами, как мы уже поступали в модели Б.

Итак, гомоморфизм – это отображение, сохраняющее свойства всех операций из области отправления (в том смысле сохраняющее, что, *во-первых*, каждой из них соответствует некоторый «гомоморфный образ» и, *во-вторых*, результат каждой из них отображается в результат применения гомоморфного образа операции к гомоморфному образу ее аргумента).

Основное свойство гомоморфизма можно представить следующей коммутативной диаграммой:

```
Результат:  f(X) -------> образ результата: g(f(X))=g(f)(g(X))
          /^\                                /^\
       f:  |                          g(f):   |
           |                                  |
   аргумент: X    ---------> образ аргумента : g(X)
```

Коммутативность диаграммы означает, что из ее левого нижнего угла в правый верхний можно пройти любым из указанных стрелками путей с одинаковым результатом.

Обычно гомоморфизм объясняют как отображение из «богатых» структур в «бедные», при котором сохраняются алгебраические законы отображаемых структур. Но для нас особенно интересна (обратная, двойственная) интерпретация гомоморфизма как отношения между «бедной» и «богатой» структурами, при котором в последней сохраняются все свойства сохраняемых из первой операций. В сущности, *это и есть отношение идеального наследования*!

Достаточно посмотреть на наш идеал наследования и на определяющее гомоморфизм соотношение, или на коммутативную диаграмму.

Действительно, исходным объектам в ее правом нижнем углу соответствуют обогащенные объекты в левом нижнем углу. Их обогащение проявляется пока лишь в том, что их просто «больше» – каждому обогащенному соответствует «бедный».

Наследовать естественно не все операции (возможности, свойства), а лишь те, которые желательно, поэтому не у всех операций из Q имеются прообразы в левой части диаграммы. Но если что-то наследуется, то результаты «богатых» операций для «бедных» должны «сохраняться» – это и отражено в коммутативности схемы, так как «сохранение» содержательно означает выполнение для прообразов в P тех же алгебраических законов, что и для их образов в Q.

Наконец, и требованию защиты от разрушения исходных услуг, сохранению работоспособности старых программ, пользующихся этими услугами (и косвенно отсутствие необходимости в перетрансляции и даже запрет на нее – иначе окажутся под угрозой старые программы!), тоже нашлось место в определяющем гомоморфизм соотношении. Ведь не предполагается какой-либо тождественности P и Q или их частей. В общем случае годится любое отображение, лишь бы выполнялось основное соотношение. В частности, допустимо и тождественное отображение компонент P и Q, когда оно устраивает.

Итак, **гомоморфизм из обогащенного типа в исходный – естественное математическое представление (математическая модель) идеального наследования**. Примеры обогащаемых и обогащенных типов приводились.

Можно добавить еще одну серию примеров. Как известно, программы на ЯП представляют текстами – последовательностями литер. Будем считать, что определен исходный тип «Строчка» – последовательность литер. Его обогащение – тип «Абстрактное_дерево». Объект этого типа представляет собой дерево с листьями из литер исходной строчки, с доступом к компонентам не только по номерам литер, но и по селекторам, выбирающим поддеревья. Новое обогащение – тип «Синтаксический_объект». Одному абстрактному дереву при этом может соответствовать много синтаксических объектов. Например, одно и то же абстрактное дерево может играть роль «строка» или «идентификатор», или даже «буква» (и получить соответствующее значение «синтаксического» атрибута).

Еще одно обогащение – тип «Вхождение_синтаксического-обьекта» – одному идентификатору соответствуют несколько его вхождений (в частности, определяющее и использующие). Обогащаются, соответственно, и атрибуты. Например, у вхождения может появиться атрибут «координата», которого не было у идентификатора. Наконец, другим обогащением синтаксического объекта может служить так называемое атрибутированное дерево, например идентификатор с атрибутами «тип», «область видимости» и др.

Упражнение. Опишите соответствующие обогащения на Обероне. Опишите также соответствующие гомоморфизмы.

Подсказка. Гомоморфизм

основа: СинтОбъект -> АбстрДерево

с сохранением операции выборка_поддерева_по_селектору, причем

основа(X.C) = основа(X).C,

где X: СинтОбъект, C: Селектор, а «.» – операция выборки.

Другие применения гомоморфизма в программировании. Важно заметить, что понятие гомоморфизма удачно моделирует не только отношение наследования типов, но и многие другие важнейшие понятия программирования. Конечно, содержательно все они связаны с тем или иным представлением о «корректном развитии», «корректном обогащении» ранее созданного, рассмотренного, изученного и т. п.

Например, **метод пошаговой детализации** можно считать методом обогащения ранее рассмотренных структур, при котором к ним оказывается применимым все более богатый набор операций. Так, начав создавать пакет в Аде пошаговой детализацией, мы сначала могли лишь переписывать фрагменты программы («переписать» – первая операция, применимая к структурам-фрагментам), затем получили возможность транслировать спецификацию пакета (вторая операция) и лишь потом исполнять программу (третья операция).

Очевидно и накопление атрибутов при пошаговой детализации. Достаточно вспомнить уже обсуждавшийся переход от текстов к абстрактным деревьям, затем к синтаксическим, затем атрибутированным.

Подчеркнем, что всегда существует тривиальный гомоморфизм из одних структур в другие, отображающий все мыслимые операции в «ничегонеделание». Мы говорим о естественном гомоморфизме, отражающем содержательное соответствие всех существенных операций.

Еще один пример гомоморфизма – **связь по импорту в Аде, Модуле-2, Обероне** и др. Импортирующий пакет служит областью отправления, а импортируемый – областью прибытия. Следующий пример: метод экспортного окна в Обероне. Область отправления – модуль, область прибытия – его спецификация. Последний из этой серии примеров – **отображение состояния, построенного в результате нахождения очередной согласующей подстановки (в реляционном программировании) на исходное состояние**. Здесь сохраняемые операции – доступ по именам к значениям.

Какую пользу можно извлечь из изложенных выше соображений?

Во-первых, факт, что казавшиеся совершенно различными понятия программирования описываются единым математическим понятием, может принести пользу математической теории ЯП.

Во-вторых, уже сейчас можно классифицировать некоторые языковые конструкты на «чистые» и «нечистые» с точки зрения идеальной развиваемости в зависимости от того, описывается ли предлагаемое ими развитие естественным гомоморфизмом. Назовем соответствующий критерий **ЕГМ-критерием**. Он работает аналогично тому, как левая факторизация (**разделенность**) грамматики служит критерием пригодности ЯП для эффективного анализа.

Конечно, и без ясного понимания или формулировки источника несообразности во многих случаях интуитивно чувствуется нарушение естественной регулярности структур или свойств. Критерий ЕГМ позволяет придать таким оценкам прочную математическую основу.

Например, *указатель сокращений «use» в Аде не согласуется с ЕГМ-критерием* (оказывается отступлением от естественного гомоморфизма). Не согласуется по-

тому, что это не чистая вставка контекста, а вставка с «фокусами», нарушающими естественное отображение имен использующего контекста на имена вставляемого.

Укажем еще один ЕГМ-фильтр: *предикаты с побочным эффектом должны быть отнесены к «нечистым» языковым конструкциям*, так как могут «поломать» уточняемое состояние и тем самым сделать невозможным естественный гомоморфизм из нового состояния в исходное.

Посредством ЕГМ-критерия легко отделить *конкретизацию-обогащение от конкретизации-специализации* (реализуемой, например, универсальным специализатором). Ведь последней позволительно «ломать» исходные структуры специализируемой программы, и в отличие от чистой конкретизации-обогащения в случае специализации может оказаться невозможным естественный гомоморфизм из новых структур в старые.

> **Упражнение** (повышенной трудности). Опишите точнее все упомянутые гомоморфизмы или обоснуйте их отсутствие.

Итак, ЕГМ-критерий может оказаться полезным при работе с ЯП, в особенности с авторской позиции. Другими словами, *получен работающий критерий качества ЯП в целом и качества отдельных его конструктов*.

Глава 7

Объектно-ориентированное программирование

7.1. Определяющая потребность .. 416
7.2. Ключевые идеи объектно-ориентированного программирования 417
7.3. Пример: обогащение сетей на Турбо Паскале 5.5 418
7.4. Виртуальные операции 423
7.5. Критерий Дейкстры 431
7.6. Объекты и классы в ЯП Симула-67 432
7.7. Перспективы, открываемые объектной ориентацией средств программирования 434
7.8. Свойства объектной ориентации 437
7.9. Критерий фундаментальности языковых концепций 438

7.1. Определяющая потребность

На протяжении всей книги мы неоднократно отмечали потенциальную пользу от введения активных данных, активных обрабатываемых (и одновременно обрабатывающих) объектов. Приводились примеры таких объектов, в частности объектов задачных типов (процессов) в Аде. Однако активные объекты всегда обладали некоторой экзотикой, всегда были лишены существенных «прав» обычных данных, а «обычные» данные – свойств активных объектов.

Например, если пакет в Аде может содержать определения разнообразных программных услуг и с этой точки зрения может считаться как пассивным, так и активным объектом, то он не обладает типом, недопустимы переменные-пакеты и т. п. Если допустим задачный тип, то в задаче нельзя непосредственно определить никаких услуг, кроме входов; объекты задачного типа нельзя присваивать (задачные типы считаются ограниченными приватными). С другой стороны, комбинированные типы в Аде не могут содержать процедур в качестве компонент.

Для ограничений такого рода находятся свои резоны, однако в целом они усложняют ЯП, а по *закону распространения сложности* – и все, что связано с ним.

Настало время освободиться от неестественных ограничений и подняться на следующий уровень абстракции – ввести *концепцию языкового объекта, обобщающую представление об активных и пассивных объектах*. Конечно, это фактически означает, что любой объект придется рассматривать как потенциально активный, если явно не оговорено обратное. Потребность в концепции подобного рода может показаться надуманной, если не учитывать, что она непосредственно связана с потребностью овладеть сложностью программирования, которую лишь частично удовлетворяют все рассмотренные ранее языковые концепции. Фактически речь идет о потребности рационально сочетать все их преимущества за счет нового уровня абстракции.

Как мы увидим, эта цель в значительной степени оказалась достижимой на современном этапе развития программирования в рамках объектно-ориентированного программирования. Выяснилось, что и другие перспективные тенденции в сочетании с ним проявляют всю свою мощь и привлекательность. Иными словами, объектно-ориентированное программирование стало катализатором нового взрыва творческой активности в области ЯП, первые результаты которого в виде практичных коммерчески доступных систем программирования уже появились. Еще более впечатляющие достижения впереди.

Выберем для определенности только один аспект обозначенной нами определяющей потребности, а именно развиваемость, и покажем путь к объектной ориентации через стремление к идеалу развиваемости, а затем отметим и иные перспективы, открываемые объектной ориентацией. Конечно, можно было взять за основу и иные аспекты определяющей потребности – наш особый интерес к развиваемости объяснен в предыдущем разделе.

7.2. Ключевые идеи объектно-ориентированного программирования

Первое фундаментальное изобретение, обеспечившее принципиальные продвижения к идеалу развиваемости, состоит в том, чтобы **сделать программный модуль нормальным языковым объектом** (в частности, объектом вполне определенного типа).

Тем самым в общем случае нормальный языковый объект становится носителем программных услуг, а управление такими объектами – управлением предоставляемыми программными услугами, в частности их развитием и защитой (ведь программный модуль – это носитель одной или нескольких программных услуг, предназначенных для совместного санкционированного использования и защищенных от использования несанкционированного).

Требуемый уровень гибкости в управлении атрибутами программных модулей (в частности, видимыми в них именами) достигнут уже в таких ЯП, как Ада или Модула-2. Обеспечена определенная гибкость и в управлении типами. Однако из-за того, что модули не считаются обычными типизированными языковыми объектами, для модулей и типов нужны два специальных способа управления, каждый со своими проблемами и, главное, с дополнительными проблемами для программиста. Унификация этих способов управления открывает путь, в частности, к ясной и гибкой концепции развиваемости.

Однако когда программный модуль становится объектом некоторого типа, он остается активным объектом, не только обрабатываемым, но и обрабатывающим – ведь он носитель программных услуг!

Второе фундаментальное изобретение основано на наблюдении, что **активный объект может предоставлять услуги и по извлечению и (или) преобразованию значений собственных атрибутов** – это совершенно естественно для унифицированной концепции объекта (и типа). Но в такой ситуации становится удобным ЛЮБУЮ программную услугу (действие, операцию, подпрограмму, ресурс) рассматривать как принадлежащую конкретному объекту некоторого типа. И управление предоставлением и развитием услуг становится столь же гибким, как управление любыми другими атрибутами объектов (или столь же негибким!).

Так мы приходим к тому, что в последние годы привлекает всеобщее внимание и получило не слишком удачное название – **объектно-ориентированное программирование**. Можно надеяться, что читателю теперь понятно происхождение этого названия (в сущности, программировать в этом стиле приходится только услуги, предоставляемые объектами тех или иных типов). Название не слишком удачное, в частности, потому, что такие активные объекты лучше бы называть «субъектами», так как они полностью «самостоятельно» определяют свое поведение.

Кстати, остался последний существенный вопрос: *определяют на основе какой информации*?

Ясно, что в мире активных объектов информация может поступать только от других объектов, обращающихся за услугами (в частности, за «услугой» принять информацию).

Так мы приходим к ***третьему фундаментальному изобретению***, характерному для объектно-ориентированного программирования, – *концепции обмена сообщениями между активными объектами, самостоятельно решающими, как именно реагировать на поступающие сообщения*, в частности трактовать ли их как запрос на предоставление услуги другим объектам или просто принять к сведению.

> **Упражнение.** Нетрудно усмотреть близость к изложенным идеям, например, концепции адовских объектов задачных типов. Самостоятельно сопоставьте эти концепции (возможно, после более подробного знакомства с объектной ориентацией). Найдите аналогии и отличия в других известных вам ЯП.

Итак, в первом приближении мы познакомились с фундаментом объектно-ориентированного программирования. Пора привести содержательный пример, демонстрирующий преимущества новой концепции.

7.3. Пример: обогащение сетей на Турбо Паскале 5.5

Напишем все тот же пример с обогащением сетей, но на этот раз на ЯП Турбо Паскаль 5.5 – первой из версий широко известной системы фирмы Борланд, в которую включены средства объектно-ориентированного программирования. Имеются в ней и средства раздельной трансляции (появившиеся начиная с версии 4.0). Было заманчиво показать объектно-ориентированное программирование сразу на примере коммерческой системы. Хотя по сравнению с Адой читатель почувствует наряду с преимуществами и определенные неудобства от возврата к ограничениям Паскаля (результат функций не может быть составным, после THEN – только простой оператор, нельзя повторять имя процедуры после ее последнего END). Имеются отличия и в концепции модульности (в частности, спецификация, начинающаяся ключевым словом INTERFACE, отделена от реализации, начинающейся ключевым словом IMPLEMENTATION, но они не выделены в отдельные трансляционные модули). Подробнее об этом говорить не будем.

```
UNIT ПараметрыСети; (* модуль с пустой реализацией *)
INTERFACE
   CONST МаксУзлов = 100;
      МаксСвязей = 8;
IMPLEMENTATION
END.

UNIT УправлениеСетями;
INTERFACE
   USES ПараметрыСети; (* импорт с доступом по коротким именам *)
   TYPE
```

```
      Узел = 1..МаксУзлов;
      ЧислоСвязей = 0..МаксСвязей;
      Перечень Связей = ARRAY [1..МаксСвязей] OF Узел;
      Связи =  RECORD
                  Число : ЧислоСвязей;
                  Узлы : Перечень Связей;
               END;
      ЗаписьОбУзле =  RECORD
                         Включен : BOOLEAN;
                         Связан : Связи;
                      END;
(* до сих пор — традиционно, но ниже следует объектный тип *)
(* в Обероне его приходилось моделировать *)
      Сети = OBJECT
      C: ARRAY [1..МаксУзлов] OF ЗаписьОбУзле;
         PROCEDURE Инициализировать;
         PROCEDURE Вставить(X : Узел);
(* у всех процедур — на один параметр меньше! *)
         PROCEDURE Удалить(X : Узел);
         PROCEDURE Связать(АУзел, ВУзел : Узел);
         PROCEDURE Присвоить(VAR Сеть : Сети);
         PROCEDURE УзелЕсть(X : Узел) : BOOLEAN;
         FUNCTION ЕстьСвязь(АУзел, ВУзел : Узел) :BOOLEAN;
         PROCEDURE ВсеСвязи(X : Узел; VAR R : Связи);
      END;
(* среди компонент объектов типа Сети — и обычные поля (данные, например поле C), и
активные компоненты (операции или правила действий, например Вставить) *)

IMPLEMENTATION
PROCEDURE Сети.Инициализация; (* такой операции не было, она и раньше была бы
полезной, а при работе с объектным типом становится обязательной *)
   VAR i: 1..МаксУзлов;
   BEGIN (* *)
      FOR i:= 1 TO МаксУзлов DO
         BEGIN
            C[i].Включен := FALSE; C[i].Связан.Число:= 0;
         END;
END;

(* В объявлениях процедур с префиксом Сети видимы все имена и объявления этого
типа, и обращаться к такой процедуре следует как к обычному полю конкретного
объекта типа Сети — примеры будут даны ниже.
Поэтому во всех операциях на один параметр меньше — не нужно передавать в качестве
параметра обрабатываемую сеть. Ведь любая операция работает именно с той конкретной
сетью, которой принадлежит. *)

PROCEDURE Сети.УзелЕсть(X : Узел) : BOOLEAN;
BEGIN
```

```
   RETURN C[X].Включен; (* доступ короче — работаем в нужной сети *)
END; (* повторять здесь названия процедур в Паскале нельзя — оцените неудобство! *)
   (* хотя, конечно, можно применять комментарии *)
PROCEDURE Сети.ВсеСвязи(X : Узел; VAR R : Связи);
BEGIN
   R := C[X].Связан;
END;
PROCEDURE Сети.Вставить(X : Узел);
BEGIN
   C[X].Включен := TRUE;
   C[X].Связан.Число := 0;
END;
PROCEDURE Сети.Присвоить(VAR Сеть : Сети);
BEGIN
   C := Сеть.C;
END;

FUNCTIONСети.ЕстьСвязь(АУзел, ВУзел : Узел): BOOLEAN;
   VAR i : ЧислоСвязей;
BEGIN
   ЕстьСвязь := FALSE;
   WITH C[АУзел].Связан DO
      FOR i := 1 TO Число DO
         IF Узлы[i] = ВУзел THEN
            ЕстьСвязь := TRUE;
END;

PROCEDURE Сети.Связать(АУзел, ВУзел : Узел);
      PROCEDURE Установить_связь(Откуда, Куда : Узел);
      BEGIN
   WITH C[Откуда] DO (* вставлен контроль *)
   IF not Включен THEN write('узел',Откуда,'не включен!');
   WITH C[Откуда].Связан DO
      BEGIN
         IF Число >= МаксСвязей THEN
            Write('В узле',Откуда,'нет места для связей');
            Число := Число+1;
            Узлы(Число) := Куда;
         END;
      END;
   BEGIN
IF not Есть_связь(АУзел, ВУзел) THEN
BEGIN
   Установить связь(АУзел, ВУзел, ВСети);
   IF АУзел < > ВУзел THEN (* "< >" — знак неравенства*)
      Установить_связь(ВУзел, АУзел);
   END;
END;
```

```
PROCEDURE Сети.Удалить(X : Узел);
   VAR i; ЧислоСвязей;
   PROCEDURE Переписать(ВУзле: Узел; После: ЧислоСвязей)
      VAR j: ЧислоСвязей;
   BEGIN
      j := После;
      WITH C[ВУзле].Связан DO
      WHILE J < Число-1 DO
         Узлы[j] :=Узлы[j+1];
         j:=j+1;
      END;
   END;
   PROCEDURE Чистить(Связь, ВУзле : Узел);
      VAR i : ЧислоСвязей;
   BEGIN
      i:=l;
      WITH C[ВУзле].Связан DO
      REPEAT
         IF Узлы[i] = Связь THEN
            BEGIN
               Переписать(ВУзле, i);
               Число := Число-1;
               EXIT;
            END; i:=i+l;
         UNTIL i < Число+1
         END;
      BEGIN
         C[X].Включен := FALSE; i := 1;
         WITH C[X].Связан DO
            BEGIN
               REPEAT
                  Чистить(X, Узлы[j]);
                  i := i+1;
               UNTIL i < Число+1;
               Число := 0;
            END;
      END;
END

PROGRAM Клиент; (* головная программа *)
USES УправлениеСетями;
   VAR Сеть1, Сеть2 : Сети; (* объявление экземпляров объектного типа *)
BEGIN
   Сеть1.Инициализировать; (* работа программы — это работа объектов *)
   Сеть2.Инициализировать;
   Сеть1.Вставить(33, 13);
   Сеть2.Присвоить(Сеть1); (* объект как параметр для другого объекта *)
END Клиент;
```

```
UNIT УправлениеСетямиСВесом;
   INTERFACE
      USES УправлениеСетями, ПараметрыСети;
   TYPE
      Вес = INTEGER;
      СетиСВесом = OBJECT(Сети)       (* обогащение аналогично Оберону *)
      A: ARRAY [1..МаксУзлов] OF BOOLEAN; (* для процедуры ВесПути *)
      B: ARRAY [1..МаксУзлов] OF Вес;
      PROCEDURE Вставить(X: Узел; P: Вес);
      PROCEDURE Присвоить(VAR Сеть : Сети);
      FUNCTION ВесПути(X,Y: Узел): Вес;
   END;
END;
IMPLEMENTATION
      PROCEDURE СетиСВесом.Вставить(X : Узел; P: Вес);
      BEGIN  (* в отличие от Оберона правила видимости *)
            C.Вставить(X); (* обеспечивают краткость и защиту других объектов *)
      END;
      PROCEDURE СетиСВесом.Присвоить (VAR Сеть: Сети);
      BEGIN
            Сети.Присвоить(Сеть); (* обращение из объекта типа СетиСВесом
к операции, находящейся в его подобъекте родительского типа Сети *)
            A := Сеть.A;
            B := Сеть.B;
      END;
(* Реализация следующей операции (функции) на Обероне не приводилась. Поэтому ее не
стоит учитывать при сопоставлении удобства программирования на Турбо Паскале 5.5 и
на Обероне. Программа приведена для полноты и в качестве решения данной ранее
задачи. *)

FUNCTION СетиСВесом.ВесПути(X,Y: Узел;): Вес;
   VAR i : Узел;
   P : Вес;
   PROCEDURE ВП(X, Y: Узел; VAR PR: Вес);
      VAR   j : ЧислоСвязей;
   BEGIN PR := -1;
      IF X = Y THEN PR := B[X]
         ELSE IF ЕстьСвязь(X, Y) THEN PR := B[X] + B[Y]
            ELSE
            BEGIN  A[X]:=FALSE;
            (* чтобы не зациклиться при поиске пути *)
            WITH C[X].Связан DO
               FOR j :=1 TO Число DO
                  IF A [Узлы[j]] THEN
                  (* рекурсивный вызов ВП *)
                     BEGIN ВП(Узлы[j], Y, PR);
                        IF PR >= 0 THEN
                        (* путь найден *)
```

```
                        BEGIN PR:=B[X]+PR; EXIT END;
                  END;
            END;
         IF not A[Y] and (PR <= 0) THEN
         BEGIN
            writeln;
            write('нет пути между',X,'и',У);
            A[Y] := TRUE;
            (* чтобы не повторять сообщений при выходе из рекурсии*)
         END;
   END;
   BEGIN
      FOR i:=l TO МаксУзлов DO A[i] := TRUE;
         A[Y] := FALSE;
         ВП(X,Y,P);
         ВесПути := P;
   END;
END.
```

Итак, действующими лицами в программе становятся активные объекты с полями-процедурами. Последние принято называть правилами (действий) объектов, **методами**, **операциями** объектного типа. Мы будем употреблять термин «операция», или «правило». В Турбо Паскале строгого запрета на доступ извне к обычным (непроцедурным) полям объектов нет, но все готово к тому, чтобы такой запрет ввести.

Естественная инкапсуляция будет полностью обеспечена – поля объектов нельзя будет испортить (доступ только через операции этого же объекта, созданные автором рассматриваемого объектного типа). В Турбо Паскале такой стиль программирования рекомендован, но не обязателен. При обогащении доступ к старым полям в Турбо Паскале допустим, но с точки зрения идеала развиваемости вполне можно было ограничиться доступом только посредством старых операций.

Приятно видеть, как программа становится компактнее и прозрачнее за счет расчистки от загромождения лишними параметрами-сетями и уточнениями. Аналогичные возможности управления видимостью действуют не только при определении, но и при использовании объектных типов. Обратите внимание, как легко разрешается потенциальный конфликт наименований, – он возможен только внутри объектов (но тогда и разбираться с ним могут в принципе сами объекты, хотя в Турбо Паскале действуют и общие правила, связанные с так называемыми **виртуальными операциями**).

7.4. Виртуальные операции

Виртуальные операции Турбо Паскаля 5.5 служат примерами абстрактных атрибутов. Их конкретизация происходит в контексте обогащенного (производного) типа и состоит в сопоставлении именам операций конкретных операций из этого контекста, обладающих теми же названиями. Виртуальные операции Турбо Пас-

каля 5.5 не допускают такой глубокой конкретизации, как атрибуты-поля, поскольку могут быть связаны только с типом, а не с конкретными экземплярами этого типа.

Один из способов реализации виртуальных операций виден из примера в Обероне, где операция (например, Присвоить) из контекста, в котором создается обогащенный объектный тип СетиСВесом, присваивается переменной, объявленной в том контексте, где объявлен исходный объектный тип (Сети). Ясно, что аналогично можно присваивать сами операции (или указатели на них) конкретным объектам (экземплярам объектного типа) в любом подходящем контексте.

В Турбо Паскале для аналогичной цели служит **Таблица Виртуальных Операций** (ТВО), которую строит компилятор для каждого объектного типа с виртуальными операциями. Каждый объект такого типа содержит указатель на ТВО. Так что программист избавлен от необходимости явно программировать соответствующие объявления процедурных переменных и присваивания, а при использовании обогащенных объектов старые операции заменяются автоматически на обновленные операции с теми же именами.

При объявлении объектных типов с виртуальными операциями (а также при объявлении их обогащений) необходимо объявлять хотя бы одну так называемую **операцию-конструктор** и хотя бы одну **операцию-деструктор**. Они выделяются ключевыми словами constructor и destructor соответственно. Первые предназначены для настройки создаваемого объекта на контекст, вторые – для удаления объекта (в частности, освобождения памяти). Конструкторы и деструкторы сами могут быть виртуальными. Первой операцией, работающей в объекте, должен быть его конструктор (один из его конструкторов), последней операцией (при аккуратном программировании) – деструктор.

Введение в ЯП конструкторов и деструкторов – попытка достичь большей ясности программы, упрощения контроля и оптимизации, сохранив высокий уровень динамизма в управлении созданием и удалением объектов со стороны программиста.

В качестве примера применения виртуальных операций приведем с краткими комментариями демонстрационные модули из фирменного руководства по системе Турбо Паскаль 5.5 (Object-Oriented Programming Guide):

```
unit Points; (* модуль "Точки" *)
interface
uses Graph;
   (* модуль, предоставляющий графические операции *)
type
   Location = object   (* объектный тип "Координата" *)
      X,Y : Integer;
      procedure Init(InitX, InitY : Integer);
      function GetX : Integer; (* функция дайХ *)
      function GetY : Integer; (* функция ДайУ *)
   end;
   Point = object(Location)
         (* объектный тип "Точка" *)
      Visible : Boolean;  (* Видимо *)
```

```
      procedure Init(InitX, InitY : Integer);
      procedure Show; (* Показать *)
      procedure Hide; (* Скрыть   *)
      function IsVisible : Boolean; (* Видимо? *)
      procedure MoveTo(NewX, NewY: Integer); (* Переместить *)
   end;
implementation
{ Реализация операций типа Location: }
Procedure Location.Init(InitX, InitY : Integer);
begin
   X := InitX;
   Y := InitY;
end;
function Location.GetX : Integer;
begin
   GetX := X;
end;
function Location.GetY : Integer;
begin
   GetY := Y;
end;

{ Реализация операций типа Points: }
procedure Point.Init(InitX, InitY : Integer);
begin
      Location.Init(InitX, InitY);
      Visible := False;
end;
procedure Point.Show;
begin
      Visible := True;
      PutPixel(X, Y, GetColor); (* услуги модуля Graph — нарисовать точку указан-
ного цвета *)
end;
procedure Point.Hide;
begin
      Visible := False;
      PutPixel(X, Y, GetBkColor);(* услуги модуля Graph — нарисовать точку фоново-
го цвета *)
end;
function Point.IsVisible : Boolean;
begin
      IsVisible := Visible;
end;
procedure Point.MoveTo(NewX, NewY : Integer);
begin
      Hide; (* параметры не нужны! Сама точка — активный объект *)
      Location.Init(NewX, NewY);
      Show;
end;
end.
```

Пока ни одной виртуальной операции нет. Так как одни операции используют другие (например, Point.Init или Point.MoveTo используют операции типа Location), то они будут использовать именно эти старые операции даже тогда, когда будут введены обогащения типов Location и Point. Если это нежелательно, типы Location и Point нужно программировать так, как показано в следующем модуле, снабжая подлежащие последующей замене операции признаком virtual.

```
unit Figures; (* модуль Фигуры *)
interface
uses Graph, Crt; (* еще один вспомогательный модуль *)
(* разбираться в том, из какого именно модуля импортированы имена, в Турбо Паскале
неприятно — оцените решение из Оберона! Полезно сопоставить и с Модулой-2. Правда,
квалифицированный программист станет систематически применять комментарии, если ЯП
не его заставляет сообщать потенциальному читателю столь необходимую информацию. *)
type
   Location = object
      X,Y : Integer;
      procedure Init(InitX, InitY : Integer);
      function GetX : Integer;
      function GetY : Integer;
   end;
   PointPtr = ^Point; (* "^" — знак указателя *)
   Point  = object(Location)
      Visible : Boolean;
      constructor Init(InitX, InitY : Integer);
      destructor Done; virtual;
      procedure Show; virtual;
      procedure Hide; virtual;
      function IsVisible : Boolean;
      procedure MoveTo(NewX, NewY : Integer);
      procedure Drag(DragBy : Integer); virtual;
(* задает относительный шаг при перемещении фигуры по экрану *)
   end;
   CirclePtr = ^Circle;
   Circle = object(Point) (* объектный тип "Окружность" *)
      Radius : Integer;
      constructor Init(InitX, InitY : Integer;
                       InitRadius : Integer);
      procedure Show; virtual;  (*показать *)
      procedure Hide; virtual;   (* скрыть *)
      procedure Expand(ExpandBy: Integer); virtual;    (* увеличить *)
      procedure Contract(ContractBy: Integer); virtual;  (* уменьшить *)
   end;
implementation
{ Реализация операций типа Location:  }
procedure Location.Init(InitX, InitY : Integer);
begin
   X := InitX;
   Y := InitY;
```

```
end;
function Location.GetX : Integer;
begin
   GetX:= X;
end;
function Location.GetY : Integer;
begin
   GetY:= Y;
end;

{ Реализация операций типа Points:   }
constructor Point.Init(InitX, InitY : Integer);
begin
      Location.Init(InitX, InitY);
      Visible := False;
end;
destructor Point.Done;
begin
   Hide;
end;
procedure Point.Show;
begin
   Visible := True;
   PutPixel(X, Y, GetColor);
end;
procedure Point.Hide;
begin
   Visible := False;
   PutPixel(X, Y, GetBkColor);
end;
function Point.IsVisible : Boolean;
begin
   IsVisible := Visible;
end;
procedure Point.MoveTo(NewX, NewY : Integer);
begin
   Hide;
   X := NewX;
   Y := NewY;
   Show;
end;
(* пока все, как было; ниже обеспечивается движение фигуры по экрану; все начинает-
ся со вспомогательной функции, проверяющей наличие изменений координат *)
function GetDelta(var DeltaX : Integer;
                  var DeltaY : Integer) : Boolean;
var
   KeyChar : Char;
   Quit : Boolean;
begin
```

```
DeltaX := 0; DeltaY := 0; { 0 означает отсутствие изменений } GetDelta := True;
repeat (* запрос изменений *)
   KeyChar:= ReadKey; { Считывается нажатие клавиши }
{ можно только догадываться, из какого модуля имя ReadKey }
   Quit := True;{ Предполагается, что она допустима }
   case Ord (KeyChar) of
      0: begin {0 — расширенный двухбайтный код}
         KeyChar := ReadKey; {Считывается второй байт кода}
         case Ord (KeyChar) of
            72: DeltaY := -1; {Клавиша Up; уменьшение Y}
            80: DeltaY := 1; {Клавиша Down; увеличение Y}
            75: DeltaX := -1;{Клавиша Left; уменьшение X}
            77: DeltaX := 1; {Клавиша Right; увеличение X}
         else Quit := False;
         {Другие коды игнорируются}
         end; { case } (* так применяются комментарии *)
      end;
     13:GetDelta:=False;{Клавиша "Исполнение" означает отсутствие
     (конец) изменений }
     else Quit .= False; { Игнорируются другие клавиши }
   end;  { case }
until Quit;
end;
procedure Point.Drag(DragBy : Integer);
var
   DeltaX, DeltaY : Integer;
   FigureX,'FigureY : Integer;
begin
   Show; { Показывается фигура, подлежащая перемещению }
   FigureX := GetX; FigureY := GetY;
   { Цикл собственно перемещения: }
   while GetDelta(DeltaX, DeltaY) do
   begin
      FigureX := FigureX + (DeltaX * DragBy);
      FigureY := FigureY + (DeltaY * DragBy);
      MoveTo(FigureX, FigureY);
   end;
end;
{ Реализация операций типа Circle: }
constructor Circle.Init(InitX, InitY : Integer;
                        InitRadius : Integer);
begin
   Point.Init(InitX, InitY);
   Radius := InitRadius;
end;
procedure Circle.Show;
begin
   Visible := True;
   Graph.Circle(X, Y, Radius); (* рисуется окружность *)
```

```
end;
procedure Circle.Hide;
var
   TempColor : Word;
begin
   TempColor := Graph. GetColor;
   Graph.SetColor(GetBkColor);
   Visible:= False;
   Graph.Circle(X, Y, Radius); (* чтобы стереть, рисуется окружность фонового цвета *)
   Graph.SetColor(TempColor);
end;
procedure Circle.Expand(ExpandBy : Integer);
begin
      Hide;
      Radius:= Radius + ExpandBy;
      if Radius <0 then Radius := 0;
      Show;
end;
procedure Circle.Contract(ContractBy : Integer);
begin
   Expand(-ContractBy);
end;
{ Раздел инициализации в этом модуле отсутствует }
end.
program FigureDemo; (* Главная программа *)
uses Crt, DOS, Graph, Figures;
type
   Arc = object(Circle) { объектный тип "Дуга"}
      Start Angle, EndAngle : Integer; (* начальный и конечный углы *)
      constructor Init(InitX, InitY : Integer;
                       InitRadius : Integer;
                       InitStartAngle, InitEndAngle : Integer);
      procedure Show; virtual; (* заменять виртуальные можно только виртуальными *)
      procedure Hide; virtual;
   end;
var
   GraphDriver : Integer;
   GraphMode : Integer;
   ErrorCode : Integer;
   AnArc : Arc; ACircle : Circle;
{ реализация операций типа Arc: }
constructor Arc.Init(InitX.InitY : Integer;
                   InitRadius : Integer;
                   InitStartAngle, InitEndAngle : Integer);
begin
   Circle.Init(InitX, InitY, InitRadius);
   StartAngle:= InitStartAngle;
   EndAngle:= InitEndAngle;
end;
```

```
procedure Arc.Show;
begin
   Visible := True;
   Graph.Arc(X, Y, StartAngle, EndAngle, Radius);
(* при работе с дугами нельзя пользоваться операциями над полными окружностями,
поэтому применяется виртуальная операция Show (Показать) *)
end;
procedure Arc.Hide;
var
   TempColor : Word;
begin
   TempColor:=Graph.GetColor; Graph.SetColor(GetBkColor);
   Visible := False;
   (* вычерчивание дуги в фоновом цвете, чтобы скрыть ее *)
   Graph.Arc(X, Y, StartAngle, EndAngle, Radius);
(* при работе с дугами нельзя пользоваться операциями над полными окружностями,
поэтому применяется виртуальная операция Hide (Скрыть) *)
   SetColor(TempColor);
end;

{ Тело главной программы: }
begin
      GraphDriver := Detect; {Используются услуги модуля DOS для определения типа
применяемой клавиатуры}
      DetectGraph(GraphDriver, GraphMode);
      InitGraph(GraphDriver, GraphMode,' ');
      if GraphResult < > GrOK then (* можно ли пользоваться графикой? *)
         begin
            WriteLn('>>Halted on graphics error:' , GraphErrorMsg(GraphDriver));
            Halt(1)
         end;
{ Все обогащения типа Point содержат виртуальные операции и поэтому перед использо-
ванием должны быть инициализированы конструкторами; ниже следует инициализация
объектов ACircle и AnArc }
      ACircle.Init(151, 82, {Начальные координаты центра — 151, 82;}
                     50);   {начальный радиус — 50 точек растра}
      AnArc.Init(151, 82,  {Начальные координаты центра — 151, 82;}
               25, 0, 90); {начальный радиус — 50 точек растра}
                           {Нач. угол: 0; Кон. угол: 90}
{ Замените AnArc на ACircle, чтобы перемещать окружности вместо дуг. Нажмите
клавишу «исполнение» (Enter), чтобы прекратить перемещения (и завершить программу) }
      AnArc.Drag(5);  { Устанавливается шаг перемещения }
(* при нажатии соответствующих клавиш-стрелок дуга (или окружность) перемещаются по
экрану, наглядно демонстрируя "активность" объектов *)
      CloseGraph;
      RestoreCRTMode;
end.
```

Итак, если бы не виртуальные операции, то не удалось бы программировать работу с дугами как обогащение работы с окружностями. Виртуальные операции,

определенные для дуг, всюду перекрывают виртуальные операции, определенные для окружностей, в том числе и в самих операциях для окружностей. Ради последнего огород и городился!

Имена непосредственных компонент объекта должны быть уникальными как в объявлении объектного типа, так и во всех его обогащениях – коллизии имен обычных полей не допускаются, а повторное использование имен операций разрешено только при замене виртуальных операций при обогащении объектного типа. (Профили всех одноименных виртуальных операций должны совпадать! Сравните с Адой и примером в Обероне.)

Тем самым становится особенно важным давать атрибутам объекта имена, отражающие их содержательные роли не только в создаваемой программе, но по возможности и *во всех мыслимых ее развитиях*, – ведь эти имена становятся «ключевыми параметрами» развития программы.

По умолчанию для операций применяется **статическое связывание** (связывание при объявлении типа и, следовательно, уже фиксированное при трансляции определяющего этот тип модуля). Для виртуальных операций применяется так называемое **отложенное** (задержанное) **связывание** с их именами. Это связывание при объявлении обогащенного типа, не завершенное при трансляции определяющего модуля для исходного типа и, следовательно, требующее завершения при трансляции или исполнении определяющего модуля для обогащенного типа, в Турбо Паскале выполняется посредством ТВО.

Заметим, что обычно атрибуты-поля не относят к абстрактным атрибутам. Однако классификация атрибутов по степени их настраиваемости (конкретные, виртуальные, поля) не только помогает в очередной раз почувствовать пользу единого взгляда на абстракцию-конкретизацию в ЯП (в частности, выделение абстракции связывания), но и предсказать появление в перспективе ЯП, где будут правила, настраиваемые как с точностью до типа, так и с точностью до экземпляра, и даже с точностью до конкретного исполнения экземпляра. Мы приходим к полностью динамическому связыванию операций, аппарат для которого уже продемонстрирован примером в Обероне.

> С другой стороны, фактически и сейчас применяется оптимизация представления объектов, состоящая в том, что в объектах, к которым нет обращений с виртуальными правилами, не помещается ссылка на ТВО. Нетрудно представить себе возможность привязки к объекту виртуальной операции не динамически через ТВО, а прямой ссылкой на тело правила, настроенного на конкретный экземпляр объекта. Такая настройка может быть оправдана многократным выигрышем в скорости за счет систематической специализации тела операции по всем правилам конкретизирующего программирования.

7.5. Критерий Дейкстры

В свое время Дейкстра, размышляя о том, какие процедуры следует выделять специальными ключевыми словами, – рекурсивные или нерекурсивные, пришел к выводу, что выделять следует ИСКЛЮЧЕНИЯ из общего правила, достойные

оптимизации (то есть процедуры НЕрекурсивные). Другими словами, если программист не указал явно, что процедура нерекурсивная, а компилятор не сумел самостоятельно распознать ее нерекурсивность при любых допустимых значениях параметров, то последний *обязан* считать ее (потенциально) рекурсивной и соответственно транслировать (возможно, менее эффективно, чем в нерекурсивном случае).

При таком решении автора ЯП, с одной стороны, **усилия программиста требуются лишь тогда, когда нужна оптимизация**, причем эти усилия требуются на формирование некоторого ЗАПРЕТА (на использование определяемого программного объекта), имеющего целью экономию машинных ресурсов.

Когда же ЯП **по умолчанию** предполагает исключение из общего правила, ориентированное на оптимизацию, а **технологически наиболее оправданный общий случай трактует как вариант, требующий специальных указаний программиста**, то это, *во-первых*, провоцирует ошибки, *во-вторых*, засоряет программу и, наконец, *в-третьих*, отвлекает внимание программиста на проблемы оптимизации от существенно более важных проблем **правильности и надежности программы**. Назовем этот критерий выбора для автора ЯП **критерием Дейкстры**. Этот критерий становится все более актуальным по мере роста цены живого труда по сравнению с ценой машинных ресурсов.

К сожалению, авторы ЯП Турбо Паскаль 5.5 не учли критерия Дейкстры (или не стали им руководствоваться), когда решили выделять ключевым словом virtual виртуальные операции, вместо того чтобы считать содержательно виртуальными любые процедуры из объявлений объектного типа, про которые не сказано явно обратное (для чего можно использовать, например, ключевое слово own). Достаточно взглянуть на объявления типов Points и Circle, которые пестрят словом virtual, чтобы усомниться в том, что авторы поступили удачно.

А если вспомнить, что программист, не написавший этого сакраментального слова, ограничивает развиваемость (а следовательно, и тиражируемость) своей программы, причем не только в угоду эффективности, но и по ошибке, которая может быть обнаружена через годы эксплуатации программы (когда потребуется, наконец, обогатить именно то ее свойство, которое оказалось зависимым от операции, не объявленной в свое время виртуальной, – кстати, *тестировать свойство развиваемости программы – особая проблема*), становится совершенно ясным, что *такое авторское решение следует признать недальновидным*. Самое неприятное – в том, что **исправить его практически невозможно** – работает принцип консерватизма языковых ниш. Оцените глубину критерия Дейкстры!

7.6. Объекты и классы в ЯП Симула-67

Уместно вспомнить здесь о Симуле-67 как первом объектно-ориентированном ЯП [36, 37]. Поразительно, сколь точно авторы этого классического ЯП угадали перспективы программирования. Нетрудно провести прямые аналогии только что рассмотренных понятий из самых современных ЯП и понятий Симулы-67. Для краткости понятия последнего будем выделять приставкой «с-».

Действительно, с-классы – это типы объектов с квазидинамическим контролем (контроль по квалификации с-ссылок, то есть типу указателей). Объект – это с-класс (возможно, со своим квазипараллельным исполнением, то есть с-открепленный). Обогащение (наследование) – это определение с-подкласса с дополнительными атрибутами. При этом старые операции применимы к новым объектам и присваивания «старым» с-ссылкам новых объектов возможно (но не наоборот) – это управляется квалификацией с-ссылок (как уже отмечено, аналог типового контроля, кстати, частично статического – динамика требуется, например, когда с-ссылка родительского класса присваивается с-ссылке на потомка, – такое может быть и корректным, если на самом деле родительская ссылка является ссылкой на потомка (имеет право)).

С-управление видимостью развито удивительно для классического ЯП, непосредственно наследовавшего блочную идеологию Алгола-60, и неплохо обслуживает объектную ориентацию языка (хотя, конечно, не учитывает модульности, которой в эталонном языке нет).

С-операции – это активные атрибуты с-объектов, и действуют они «в рамках с-объектов» (то есть сами доступны (из других объектов!) – только через ссылку на с-объект). При этом аргументом операции может служить любая компонента использующего эту операцию контекста (в соответствии со спецификацией ее параметров). Если бы еще запретить прямой доступ к «пассивным» атрибутам с-объектов, получился бы чистый аппарат для реализации абстрактных типов данных (атрибуты-операции с-класса – это и есть операции соответствующего абстрактного типа данных). Однако в реальной Симуле-67 такие «абстрактные типы» не защищены от разрушения.

Виртуальные операции – это почти в точности с-виртуальные операции. Причем и авторы Симулы-67 не учли критерия Дейкстры – с-виртуальные операции нужно выделять словом virtual.

С-дистанционные идентификаторы аналогичны обычным выборкам по селекторам. Интересно подчеркнуть, что чем шире область возможных значений ссылочной переменной (в соответствии с ее квалификацией, то есть типом), тем меньше атрибутов с ее помощью можно указать – это естественное следствие иерархии (обогащения) объектов. Однако в Симуле-67 можно снять запрет на обращение к атрибутам подклассов из надкласса (из бедного к обогащенному, из родительского к потомку) за счет явной разрешающей «оперативной» квалификации «qua»:

```
Родитель qua потомок.атрибут_потомка
```

Такое может понадобиться, когда фрагмент программы «выпадает из иерархии» и его проще всего поместить в родительский класс (например, для организации в нем взаимодействия между объектами-потомками из разных классов).

Другими словами, *если нельзя, но очень хочется, то можно*, но при этом нужно явно сообщить транслятору о сознательном нарушении запрета. В подобном стиле действуют и авторы других «строгих» ЯП. Например, в Аде можно обойти контроль типов, применив «фиктивное» преобразование типов посредством специально для этой цели предназначенной родовой функции UNCHECKED_CONVERSION.

Вместе с тем Симула-67 на практике не смогла конкурировать с ЯП, не содержавшими столь перспективных идей, хотя и была вполне справедливо представлена авторами как «универсальный язык программирования» [36]. Не говоря уж о том, что этот ЯП явно опередил свое время, – масса программистов оказалась просто не готова воспринимать его ценности. По-видимому, важнейшим его недостатком оказалась относительно низкая эффективность исполняемых программ.

Дело в том, что в Симуле-67 почти все интерпретируется (это же характерно и для Смолтока), а не компилируется, как в Турбо Паскале 5.5. В частности, нет статического создания объектов – они создаются динамически генераторами, нет статического контроля объектных типов (то есть квалификации с-ссылок – в общем случае она контролируется динамически). Для относительной неудачи Симулы-67 сыграло свою роль и полное отсутствие средств модуляризации (раздельной компиляции) на уровне эталонного языка. В более современных реализациях они, конечно, имеются (в частности, на отечественных компьютерах БЭСМ-6 и ЕС [37]). С этой точки зрения, ЯП Симула-67 унаследовал важнейший недостаток своего непосредственного предшественника и подмножества – Алгола-60, проигравшего Фортрану прежде всего из-за отсутствия модулей.

Кроме того, нет разделения спецификации и реализации. Как уже отмечалось, нет защиты от несанкционированного доступа – контролируется только квалификация ссылок, которой программист всегда может управлять, зная структуру программы (а не знать ее не может, так как при отсутствии разделения спецификации и реализации, а также (в общем случае) раздельной компиляции вся она нужна для его работы). С другой стороны, авторы [37] утверждают, что можно иметь атрибуты, доступные только через соответствующие процедуры. Остается неясным, каким способом (в приведенном ими объяснении примера в [37, стр. 33–34] имеются противоречия).

7.7. Перспективы, открываемые объектной ориентацией средств программирования

Хотя мы рассмотрели далеко не все заслуживающие внимания примеры современных ЯП (а также аспекты) объектной ориентации, накоплено достаточно материала для обсуждения открываемых ею перспектив. Среди таких ЯП нужно назвать, по крайней мере, Смолток (в особенности Смолток/5 286) и Си++, а среди аспектов – переход от явного вызова операций к обмену сообщениями между объектами.

Будем считать последнюю идею понятной без специальных пояснений – достаточно представить себе, что каждый объект снабжен анализатором сообщений, вызывающим при необходимости соответствующую операцию этого объекта. Другими словами, каждый объект снабжен мини-транслятором сообщений, устроенным, например, по принципу синтаксического управления. Конечно, в общем

случае такой подход требует значительных затрат на анализ сообщений в период исполнения программы. Однако при определенных ограничениях на класс сообщений возможна весьма глубокая оптимизация (в перспективе с учетом конкретизирующего программирования). Во всяком случае, прямой вызов операций по статически известным именам – частный случай обмена сообщениями.

По убеждению автора, объектная ориентация знаменует и стимулирует принципиально новый уровень развития средств программирования (ЯП, в частности) потому, что позволяет естественно сочетать практически все рассмотренные нами (и некоторые иные) перспективные тенденции, тем самым создавая почву и для следующего витка развития (полезно в этой связи обратить внимание, например, на идеологию ЯП Оккам-2 с его асинхронными процессами-объектами и каналами для обмена сообщениями), а также на отечественный язык НУТ [38] с его изящным соединением объектно-ориентированного, реляционного и концептуального программирования. Рассмотрим коротко представляющиеся наиболее интересными проблемы вместе с идеями их решений в рамках объектно-ориентированного программирования. Для краткости его понятия и решения будем предварять префиксом «о-».

Проблема управления

Основной о-ответ – децентрализация управления. Вместо представления о едином исполнителе, выполняющем единую программу, создаваемую единым во многих лицах «богом»-программистом, предлагается мыслить в терминах коллектива (коллективов) взаимодействующих объектов-исполнителей, «живущих» в значительной степени самостоятельно (с точностью до о-взаимодействия) в соответствии со своими собственными правилами поведения и «о-социальными» ролями. Создание, то есть программирование, такого о-общества также следует представлять как довольно демократическую скорее историю, чем процедуру, существенно использующую развиваемость о-индивидов (как типов, так и объектов), а также относительно локальные договоренности о конкретных способах взаимодействия. Такую тенденцию можно обозначить метафорой «от автархии к анархии», в связи с чем рассматриваемый стиль программирования можно назвать «анархо-ориентированным», если в соответствии с современными воззрениями снять с понятия «анархия» привкус априорного неприятия.

Проблема взаимодействия

Основной о-ответ (в перспективе) – относительно локальное взаимопонимание на основе взаимоприемлемого языка сообщений, совершенно не обязательно единого и понятного для всех. Более того, глубина понимания конкретных сообщений участниками взаимодействия также может легко варьироваться в зависимости от их роли в решении совместных задач.

Проблема ресурсов

Основной о-ответ – разнообразие как самих ресурсов, так и способов их создания, предоставления и изъятия по соответствующим операциям-запросам-сообщениям соответствующими объектами. В принципе, в эту схему укладываются любые мыслимые варианты и их оптимизации.

Проблема развития

Основной о-ответ – идеал наследуемости. Однако в общей перспективе его следует дополнить **динамизмом** (вплоть до построения обогащенных объектов при работе других объектов), **сближением понятия модуля с понятием объекта** (объектного типа, класса), а также **наследуемостью в языке обмена сообщениями**.

Проблема защиты

И здесь основной о-ответ – идеал наследуемости, дополненный динамизмом при контроле корректности сообщений, а также в общем случае принципиальной невозможностью разрушить объект, если соответствующий приказ-сообщение не входит в согласованный язык сообщений.

Проблема классификации (типизации)

Основная метафора о-ответа – **тип = язык**. Эту метафору можно раскрыть и так, что типизация охватывает любые языковые конструкты (данные, операции, их сочетания – это путь к универсальному конструктиву типа [28]), и так, что средства описания типа тесно переплетаются со средствами определения полного языка (ведь при определении типа объектов нужно определять и воспринимаемый ими язык сообщений). Одно из «экстремистских», но не лишенных смысла толкований – к одному типу относятся объекты, «понимающие» определенный язык (или подъязык), обеспечивающий взаимодействие. [Так недалеко и до объектной нации.]

Проблема модульности

Общий о-ответ – **модуль = объект** (объектный тип). Существующие различия между этими понятиями связаны с особой ролью трансляции в жизненном цикле программы. Необходимость анализа и интерпретации сообщений в качестве аспектов функционирования объектов превращает трансляцию в одну из рядовых операций и тем самым сближает логическую структуру программы с ее физической структурой.

Проблема свободы

Проблема свободы и ответственности естественно возникает перед каждой творческой личностью, в том числе (и в весьма острой форме) – перед программистом. Поскольку объектная ориентация – специфический стиль программистского мышления, а также определенная совокупность средств программирования, предоставляемая ЯП, интересно понять, какие ответы возможны в ее рамках.

Известно, что свобода хороша до тех пор, пока она не ограничивает свободу индивида, претендующего на тот же уровень свободы. С этой точки зрения основной о-ответ – **единственным источником любых ограничений на свободу поведения объекта служит требование *взаимопонимания* (корректного обмена сообщениями) со всеми, кто ему самому необходим (для выполнения о-социальной роли)**.

Частным случаем такого требования служат и ресурсные ограничения, поскольку в общем случае ресурсы по требованию объекта предоставляются ему другими объектами. Другой частный случай – описание характера обмена сооб-

щениями-операциями в спецификации и полная свобода реализации при условии воплощения требований спецификации.

Таким образом, объектная ориентация действительно в максимальной степени способствует свободному сочетанию самых разнообразных подходов к программированию отдельных объектов (объектных типов), требуя в общем случае лишь относительно локальных соглашений о необходимом «взаимопонимании» передаваемых сообщений.

Проблема ответственности

Основной о-ответ – **полная защита от несанкционированного (языком сообщений) вмешательства в поведение объекта**, в результате чего его создатель получает возможность полностью отвечать за корректность его поведения. Другими словами, никто не может заставить объект сделать то (или сделать с ним то), чего объект не «понимает» и (или) не «контролирует» (ведь любое сообщение в общем случае анализируется самим объектом).

7.8. Свойства объектной ориентации

Новый уровень абстракции

Объектная ориентация ведет за счет нового уровня абстракции к обновлению фундаментальных концепций ЯП – управления, развития, защиты, классификации, модульности, динамизма (высокоразвитой типизацией), параллелизма (наследуемостью), спецификации, реализации, жизненного цикла программы (расслоенным программированием и др.) – и в целом к сближению проблематики ЯП с проблематикой представления знаний и искусственного интеллекта.

Например, очевидно сходство используемого в объектно-ориентированном подходе понятия объекта с понятием фрейма – одним из основных в современном представлении знаний. Понятие объекта можно считать одним из воплощений (своего рода конкретизацией) понятия фрейма. Стереотипная ситуация – это объектный тип, слоты – это поля и (или) операции (правила поведения), играющие вполне определенную роль в рассматриваемой стереотипной ситуации, конкретная ситуация – это экземпляр объекта.

Интеграция понятий и средств информатики

Объектно-ориентированное программирование знаменует очередной этап сближения (интеграции) понятий и средств информатики, характерный для нее в последние годы и проявляющийся не только в названных областях, но и в создании интегрированных сред (вспомните о назначении ЯП Оберон), в сближении ЯП и СУБД (экспортное окно – аналог концептуальной схемы в СУБД, реляционный ЯП близок к языку запросов реляционной БД), ЯП и языков логического программирования (вспомните о родстве Рефала с Прологом) и др.

Целостность ЯП

Наш анализ в очередной раз демонстрирует, что ЯП – целостная система. Затронув лишь одно его свойство – развиваемость, мы на основе принципа концеп-

туальной целостности «вытащили» новый взгляд почти на все аспекты ЯП. Если бы не уже отмеченный естественный консерватизм языковых ниш, ЯП уже могли бы стать совершенно иными. Искусство авторов новейших ЯП «объектной» ориентации проявилось, в частности, в том, что такие ЯП, как Оберон, Турбо Паскаль 5.5 или Си++, оказались внешне очень похожими на своих более традиционных предшественников и вместе с тем во всех отношениях плавно вводящими пользователей в круг совершенно новых идей (в отличие от Симулы-67 и тем более Смолтока, где к тому же за эти весьма прогрессивные идеи нужно было платить резким падением эффективности программ). С этим, возможно, в основном и связан их меньший успех у пользователей, хотя немалую роль сыграла и неготовность программистской общественности к новой системе ценностей в программировании, провозглашающей самым дорогим ресурсом труд квалифицированного человека, а не, например, время работы или память компьютера.

7.9. Критерий фундаментальности языковых концепций

Судьба объектной ориентации (от неприятия ее при появлении в Симуле-67 до современного бума) на весьма нетривиальном примере подтверждает один из основных тезисов нашей книги: ***почти все фундаментальные концепции программирования (и современных ЯП) можно объяснить, не привлекая реализаторской позиции.***

Другими словами, если необходимо привлекать реализаторскую позицию, то концепция не фундаментальна. Это, конечно, не умаляет исключительной значимости применения наилучших алгоритмов и учета всех возможностей среды для коммерческого успеха программы.

Действительно, никакая проблема реализации не мешала еще двадцать лет назад изготовить систему, по объектно-ориентированным возможностям сопоставимую с Турбо Паскалем 5.5 (то есть включить их, а также соответствующие модульные средства, еще в первые версии Паскаля). Но *само программирование должно было созреть до понимания фундаментальной значимости удовлетворения потребности в развиваемости.*

Глава 8

Заключительные замечания

8.1. Реализаторская позиция 440
8.2. Классификация языков программирования 448
8.3. Тенденции развития ЯП 451

8.1. Реализаторская позиция

В самом начале книги (стр. 29) мы выделили пять позиций, с которых намеревались рассмотреть ЯП. До сих пор реализаторской позиции уделялось мало внимания. Настало время и нам несколько подробнее поговорить о реализации ЯП.

> Безусловно, возможности и требования реализации оказывают существенное влияние на свойства ЯП. Долгое время это влияние считалось (а в значительной степени и было) определяющим. С ростом возможностей аппаратуры и методов трансляции оно ослабевает, уступая технологической позиции. Как уже сказано, основной методический тезис книги состоит в том, что подавляющее большинство свойств современных ЯП можно достаточно убедительно объяснить, не прибегая к реализаторской позиции.
>
> С другой стороны, о реализации ЯП написано много полезных книг (с точки зрения общих потребностей программистов, вполне достаточно книги [39]). Поэтому постараемся уделить внимание лишь тем аспектам реализаторской позиции, которые в доступной литературе освещены недостаточно.

Напомним роль реализатора во взаимодействии с представителями остальных выделенных нами позиций. Реализатор призван *обеспечить эксплуатацию ЯП на всех технологических этапах, опираясь на замысел автора*.

Такое понимание роли реализатора (и реализации) ЯП не стало, к сожалению, общепринятым. Иногда еще приходится бороться с устаревшей точкой зрения, что задача реализатора – обеспечить ЯП исполнителем (языковым процессором, транслятором), и только. Именно такая узкая «реализаторская позиция» (имеющая глубокие корни) – одна из причин положения, при котором мы вынуждены пользоваться ненадежными трансляторами, колдовать над невразумительными диагностическими сообщениями, страдать от произвольных изменений ЯП, отсутствия сервиса, помогающего создавать и сопровождать программы, низкого уровня учебников, отсутствия методических материалов и т. п.

Нам не удастся рассмотреть задачу реализатора во всей ее полноте достаточно подробно. Поэтому поступим так же, как в случае технологической позиции. Как вы помните, мы кратко рассмотрели жизненный цикл изделия в целом, а затем выделили только проектирование как представительный этап этого цикла. Аналогичным образом дадим общее представление о задаче реализации ЯП в целом, а затем выделим лишь один аспект реализации и займемся только им.

Итак, будем считать, что реализация в целом должна обеспечить эксплуатацию ЯП **на всех этапах жизненного цикла комплексного программного продукта (ЖЦКПП)**. Рассмотрим три этапа (стадии) жизненного цикла – проектирование, эксплуатацию и сопровождение продукта. Их достаточно, чтобы выделить важнейшие компоненты реализации.

8.1.1. Компоненты реализации

Будем исходить из того, что авторское определение ЯП имеется (для базового языка индустриального программирования в настоящее время это обычно отрас-

левой, национальный или международный стандарт; в других случаях определение ЯП может иметь менее высокий официальный статус). К авторскому определению предъявляются исключительно высокие требования. Их серьезное обсуждение выходит за рамки книги. Но одно из таких требований стоит сформулировать.

Авторское определение в идеале должно исчерпывающим образом фиксировать синтаксис и семантику ЯП. Другими словами, оно должно быть способно служить **единственным** источником сведений о допустимых языковых конструктах и их смысле. Поэтому можно ожидать (и опыт уверенно подтверждает), что авторское определение непригодно в качестве методического материала (а тем более учебника) по созданию программ на этом языке. Точно, понятно и полно описать ЯП – столь непростая задача, что не стоит ее усложнять погоней за двумя зайцами.

Рассмотрим требования к реализации с точки зрения последовательных этапов ЖЦКПП.

Реализация с точки зрения этапа проектирования программы. Чтобы обеспечить эксплуатацию ЯП на этапе проектирования программы, *требуется скорее методический материал, чем авторское определение*. Нужда в нем особенно очевидна в случае базового языка индустриального программирования, ориентированного на массовое применение. Недаром в случае с Адой первые учебники появились практически одновременно с официальным определением языка (среди них – уже упоминавшийся учебник Вегнера [18]). Так что первая важнейшая компонента реализации, необходимая в особенности при проектировании программы, – это методическое руководство (**учебник**) по программированию на рассматриваемом ЯП. Конечно, учебником не исчерпываются все потребности этапа проектирования, которые призвана удовлетворять квалифицированная реализация.

В последние годы появились, в частности, программные средства, поддерживающие пошаговую детализацию, проверку правильности, создание тестов, управление проектом и другие элементы проектирования.

Реализация с точки зрения этапа эксплуатации. Сразу ясно, что *здесь не обойтись без исполнителя соответствующего ЯП*. Причем не абстрактного, а вполне конкретного, материального, обладающего достаточными физическими ресурсами и приемлемыми характеристиками эффективности. Как известно, в настоящее время исполнители для ЯП представляют собой комплекс аппаратуры и программных продуктов, называемых **трансляторами**. Будем считать, что создание аппаратуры выходит за рамки задач, связанных с реализацией конкретного ЯП (хотя имеется тенденция к изменению этого положения). Тогда в качестве второй важнейшей компоненты реализации выделим транслятор – без него невозможно обеспечить этап эксплуатации программы. Ясно, что все потребности и этого этапа не исчерпываются транслятором.

В частности, нужна операционная система, обеспечивающая нормальное функционирование аппаратуры, нужен резидент, обеспечивающий нормальное выполнение целевой программы, и т. п.

Реализация с точки зрения этапа сопровождения. Анализируя этап сопровождения, обратим внимание на основную технологическую потребность этого этапа – *корректировать программу с минимальным риском внести ошибки*. Читатель, конечно, знаком со средствами редактирования текстов (редакторами), позволяющими вносить изменения в исходные программы. Риск ошибиться уменьшается, если редактор «знает» ЯП и позволяет вносить исправления в терминах ЯП: например, такому языковому редактору можно дать задание «в процедуре Р заменить формальный параметр А на В».

Сравните указание обычному редактору «заменить А на В» и соответствующий риск заменить «не то» А. Итак, третьей важнейшей компонентой квалифицированной реализации служит языковый **редактор**.

Совсем хорошо было бы вносить исправления не в терминах ЯП, а в терминах решаемой задачи (тогда редактор должен был бы «знать» и ЯП, и ПО, и задачу), но это – дело будущего.

Итак, беглого взгляда на три этапа жизненного цикла программы хватило для выделения трех важнейших компонент реализации ЯП: учебника, транслятора и редактора.

Другие компоненты реализации. Ограничимся только компонентами, непосредственно связанными с ЯП, считая, что реализация погружена в некоторую многоязыковую систему программирования, предоставляющую необходимые общесистемные услуги, если они не определены в ЯП (базу данных, связь с другими языками, фонды готовых программ, документов и т. п.).

Укажем этапы жизненного цикла, где применение названных компонент особенно целесообразно (хотя очевидно, что они полезны и для других этапов, в том числе и выпавших из нашего рассмотрения).

Этап проектирования – процессоры, помогающие готовить тексты исходных программ. Примерами могут служить уже упомянутые препроцессоры, поддерживающие метод пошаговой детализации программ, «знающие» определенный ЯП. Они в состоянии воспринять запись шагов детализации и выдать текст законченной (или еще не законченной) программы, попутно контролируя его правильность (в диалоговом режиме, если нужно). Полезны процессоры, позволяющие писать на структурных расширениях Фортрана, Кобола, ПЛ/1 и других «заслуженных» ЯП. Еще один класс компонент реализации – **отладчики**.

Этап эксплуатации – средства контроля и измерений как программ, так и трансляторов. Это комплект тестов, проверяющих соответствие исполнителя определению языка, **оптимизаторы** и **конкретизаторы**, настраивающие программы на конкретные условия эксплуатации.

Этап сопровождения – уже упоминавшиеся измерительные средства; средства для отслеживания и контроля изменений (**версий**); контролеры программ, проверяющие соответствие стандартам (это особенно важно для переноса программ в другую среду) или выявляющие особо ресурсоемкие места.

Кроме того, следует понимать, что развитая реализация может содержать учебники для различных категорий пользователей и программных сред, транс-

ляторы с различными предпочтительными режимами эксплуатации (особо быстрый, особо надежный, особо оптимизирующий), для различных компьютеров или программных сред, языковые редакторы с различными уровнями «интеллекта» и т. п.

Итак, будем считать достаточно обоснованным следующий тезис: *квалифицированная реализация ЯП – дело сложное, дорогое, длительное и многоплановое* (для «живого» ЯП – даже не ограниченное по времени). От качества реализации в этом широком смысле слова зависят «потребительские свойства» ЯП. Реализация – один из наиболее очевидных аспектов, переводящих понятие «язык программирования» из категории научно-технической в социальную.

> Дополнительную яркую социальную окраску этому понятию придают пользователи ЯП, иногда официально объединенные в ассоциации. Так что ЯП, тем более базовый язык индустриального программирования в наше время – явление социальное и именно такого подхода к себе требует.

На этом закончим разговор о реализации в целом. Сконцентрируем внимание на более традиционной ее части – трансляторах, точнее компиляторах.

8.1.2. Компиляторы

Компилятором называется программный продукт, предназначенный для перевода с исходного ЯП на целевой (объектный) язык (обычно – язык загрузки или иной язык, близкий к машинному).

Если для целевого ЯП исполнитель имеется, то компилятор дает возможность выполнять исходные программы в два этапа. На первом этапе – этапе **компиляции** (трансляции) – исходная программа переводится компилятором на целевой язык; на втором – этапе **исполнения** – исполнителем целевого ЯП выполняется переведенная (**целевая**) программа.

Современные языки индустриального программирования ориентируются прежде всего на технологические потребности пользователей и поэтому довольно сильно отличаются от наиболее распространенных машинных языков. Вместе с тем, как мы видели, в них многое сделано для того, чтобы можно было позаботиться о надежности и эффективности целевых программ еще на этапе компиляции (вспомните квазистатический аппарат прогнозирования-контроля). По этим двум причинам компиляторы (а не, например, интерпретаторы) служат обязательными компонентами реализации практически всех языков индустриального программирования.

Мы не стремимся дать исчерпывающее определение компилятора. Дело в том, что это понятие – скорее инженерное, чем математическое. Во всяком случае, хороший компилятор должен не только «переводить», но и сообщать об ошибках, и накапливать статистические сведения об обработанных программах, и оптимизировать свою работу с учетом особенностей потока программ. Возможны и иные требования (гибкое управление свойствами целевой программы, трассировкой, печатью листинга, диагностическими режимами и прочее).

Создать компилятор – дело очень непростое. Высококачественный компилятор с современного ЯП требует нескольких лет работы и может содержать сотни тысяч команд. При этом не случайно не названо количество требуемых специалистов. Несколько лет нужно независимо от того, можно ли выделить на это 10 или 200 человек. Близкая к оптимальной – группа из 5–15 человек.

Увеличение группы только удлинит сроки или приведет к полному краху (**закон Брукса** [40]), если не удастся найти для новых людей совершенно независимой работы (такой, например, как создание комплекта тестов, проверяющих качество компилятора).

Технологии создания компиляторов посвящена огромная литература. Выделены важнейшие технологические этапы, основные компоненты компилятора, предложены многочисленные методы реализации отдельных компонент, имеются автоматизированные системы, предназначенные для создания компиляторов.

Их успешно применяют в относительно простых случаях, когда сами переводы не слишком сложны и к ресурсоемкости компиляторов не предъявляют жестких требований. В таких условиях два-три специалиста с помощью соответствующей инструментальной системы могут изготовить компилятор примерно за месяц интенсивной работы.

Однако ЯП развиваются, требования к качеству реализации повышаются, возможности аппаратуры растут. В результате разработка компиляторов для языков индустриального программирования продолжает требовать значительных творческих усилий (правда, теперь чаще приходится не столько изобретать методы компиляции, сколько квалифицированно выбирать из имеющегося арсенала).

Полноценные учебники по созданию компиляторов еще ждут своих авторов. Много интересного и полезного на эту тему можно найти в книгах [41, 42].

8.1.3. Основная функция компилятора

Рассмотрим лишь одну, выделяемую традиционно, функцию компилятора – **строить целевую программу**. Выделяется она потому, что лучше других отражает специфику компилятора и соответствует его основному назначению. Однако и остальные функции компилятора в определенных условиях могут оказаться исключительно важными. Например, для студентов важнейшей может оказаться диагностическая функция, то есть способность компилятора помогать отлаживать программу.

Итак, будем считать, что основная задача компилятора – *перевести программу с исходного языка на целевой.*

Обозначим через LL1 исходный язык, а через LL2 – целевой язык для планируемого компилятора. Пусть L1 – множество текстов, допустимых в LL1 (то есть определяемых синтаксисом LL1), а L2 – множество текстов, допустимых в LL2.

Переводом (проекцией) с языка LL1 на язык LL2 называется отношение p из L1 в L2, то есть подмножество декартова произведения L1 * L2.

Легко догадаться, что всякий компилятор характеризуется единственной проекцией (обратное неверно!). Этой проекции принадлежат те и только те пары

(t1 , t2),

где t1 – из Ll, t2 – из L2, для которых t2 может быть получен в результате применения компилятора к t1.

Данное выше определение проекции в виде отношения подчеркивает факт, что компилятор может переводить не все тексты из L1 (например, для слишком длинных текстов может не хватить ресурсов), переводить различные тексты в один (например, если они обозначают одно и то же), переводить один и тот же текст поразному (например, в зависимости от режима трансляции – с оптимизацией или без нее).

Данное определение проекции выглядит совершенно симметричным относительно языков LL1 и LL2, хотя они содержательно играют различные роли. Чтобы подчеркнуть эти роли, иногда говорят, что проекция – частичное многозначное отображение p : L1 -> L2.

8.1.4. Три принципа создания компиляторов

Небольшой опыт по созданию компилятора у нас уже есть. В модели МТ мы практиковались в создании компилятора с языка обычных (инфиксных) выражений в польскую инверсную запись (в язык постфиксных выражений). Наш компилятор представлял собой программу из четырех предложений:

```
{ el + e2 } R -> { el } { e2 } +.
{ el * e2 } R -> { el } { e2 } *.
{ ( e ) } -> { e }.
{e}->e.
```

Проекция p, соответствующая этому компилятору, должна удовлетворять определяющему соотношению

p(F1 op F2) = p(F1) p(F2) op,

где F1, F2 – правильные инфиксные формулы, op – операция.

Уже на примере такого простого компилятора можно продемонстрировать три важных положения.

Начнем с того, что созданию компилятора должна предшествовать разработка связанной с ним проекции. Это не обязательно означает, что проекция полностью фиксируется до начала программирования компилятора. Но, во всяком случае, ее замысел предшествует разработке соответствующего алгоритма перевода и последовательно уточняется, определяя всю разработку компилятора. Например, четыре предложения нашего компилятора не могли бы быть написаны, если бы мы фактически не «держали в голове» приведенное определяющее соотношение для проекции.

Проекционный принцип. Указанные выше соображения можно оформить в виде **проекционного принципа** [43]: создание компилятора можно разбить на два технологически независимых этапа – **П-этап**, или этап разработки проекции, и **А-этап**, или этап алгоритмизации проекции. Полученное на П-этапе описание проекции (например, в виде системы определяющих соотношений) может слу-

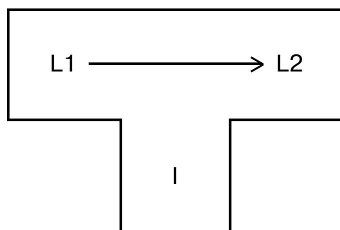
жить техническим заданием (спецификацией) для работы на А-этапе. Опыт показывает, что в некоторых практически важных случаях А-этап удается полностью автоматизировать. Делаются попытки частично автоматизировать и П-этап. Это удается за счет предварительного формального описания как исходного, так и целевого языка сходным образом на одном и том же метаязыке. В отличие от БНФ, такой метаязык должен позволять описывать не только синтаксис, но и семантику ЯП. В сущности, при этом приходится создавать проекцию описываемого ЯП на метаязык. Так что П-этап всегда носит творческий характер.

К реальным языкам индустриального программирования автоматизация П-этапа пока неприменима из-за непрактичности метаязыков и соответствующих систем построения трансляторов (СПТ).

Принцип вспомогательных переводов. Когда проекция достаточно проработана и можно приступать к ее алгоритмизации, полезно выделить две фазы компиляции – **фазу анализа** и **фазу синтеза**. В нашем примере мы воплощали первую фазу левой частью МТ-предложения, вторую – правой.

При этом результаты первой фазы представляются на некотором промежуточном языке, так что и анализ, и синтез иногда оказывается полезным, в свою очередь, считать трансляцией (соответственно, с исходного языка на промежуточный и с промежуточного на целевой). С другой стороны, в отличие от исходного и целевого ЯП, язык МТ выступает в нашем компиляторе в роли еще одного вспомогательного ЯП (**инструментального ЯП**, то есть языка, на котором написан компилятор).

Сказанное подчеркивает рекурсивную природу трансляции и может быть выделено как **принцип вспомогательных переводов**: *транслятор можно построить из трансляторов для вспомогательных языков*. Это наблюдение широко используется в различных методах и приемах создания трансляторов. Тройственную связь исходного, целевого и инструментального ЯП удобно изображать Т-образной диаграммой



С ее помощью легко описываются довольно сложные процессы, связанные с жизненным циклом компилятора (в частности, так называемая **раскрутка**, активно использующая вспомогательные переводы и применяемая при переносе компиляторов в новую программную среду).

Принцип синтаксического управления (структурной индукции). Анализ и синтез далеко не всегда удается столь четко сопоставить некоторому определенному конструкту инструментального ЯП, как это сделано в нашем простом примере.

Дело в том, что в языке МТ непосредственными составляющими при анализе выражения могут быть только выражения, термы и символы. При компиляции с более сложных исходных ЯП приходится переводить операторы, объявления, области действия и т. п. Анализ исходного текста и синтез соответствующего целевого не удается представить в этих случаях одним предложением. И для анализа, и для синтеза пишут специальные подпрограммы.

В первых компиляторах взаимодействие таких подпрограмм было довольно запутанным. Но уже в начале 60-х годов Айронсом был предложен принцип упорядочивания этого взаимодействия на основе иерархической структуры исходных текстов. Структура эта задается синтаксисом исходного языка, поэтому сам принцип получил название **принципа синтаксического управления** трансляцией (компиляцией, в частности).

В синтаксически управляемых компиляторах синтаксическим категориям исходного языка ставятся в соответствие так называемые **семантические действия**. Они-то и синтезируют целевой текст в процессе так называемой **структурной индукции**.

В этом процессе семантические действия, соответствующие определенным синтаксическим категориям, используют результаты семантических действий, соответствующих непосредственным компонентам этих категорий. Структура, по которой ведется индукция, строится в процессе анализа (декомпозиции) исходного текста в соответствии с определением исходного ЯП.

Принцип синтаксического управления и структурную индукцию можно в первом приближении понять на примере нашего компилятора для перевода выражений.

При этом левые части МТ-предложений выполняют декомпозицию (выделяя сумму, произведение, скобочную первичную формулу), а правые части – структурную индукцию, пользуясь уже готовыми переводами компонент соответствующих синтаксических категорий.

В нашем компиляторе анализ и синтез чередуются (компилятор однопроходный). Но можно сначала полностью проанализировать исходный текст, получив в результате его структуру (обычно в виде **синтаксического дерева** – это вариант промежуточного языка), а затем (на втором проходе) выполнить структурную индукцию. Иногда применяют и большее число проходов (обычно при ограничениях на память для размещения компилятора или при необходимости оптимизировать программу).

Итак, мы выделили один из принципов, позволяющий структурировать процесс создания транслятора, – *проекционный принцип*; один из принципов, позволяющих структурировать сам компилятор, – *принцип вспомогательных переводов*, и один из принципов, позволяющих структурировать синтез, – *принцип синтаксического управления* (несколько упрощая, можно отождествить его с принципом структурной индукции).

Отметим, что термин «структурная индукция» обозначает также один из способов доказательства свойств структурированных объектов.

Подчеркнем, что сам ЯП несравненно стабильнее (консервативнее), чем аппаратура и методика реализации. С другой стороны, последняя авторская реализация Модулы-2 выполнена оправдавшими себя методами двадцатилетней давности – еще одно подтверждение принципа чайника. В сущности, лишь вопрос о *возможности или невозможности реализации* в современных условиях кардинально влияет на грамотно проектируемый ЯП. В остальном влияние реализаторской позиции обычно преувеличивают.

8.2. Классификация языков программирования

8.2.1. Традиционная классификация

Изучение ЯП часто начинают с их классификации. Различают ЯП низкого, высокого и сверхвысокого уровней; процедурные и непроцедурные, диалоговые и пакетные; вычислительной, коммерческой, символьной ориентации; выделяют ЯП системного программирования, реального времени, «параллельные» ЯП; даже классические, новые и новейшие.

Определяющие факторы классификации обычно жестко не фиксируются. Чтобы создать у читателя представление о характере типичной классификации, опишем наиболее часто применяемые факторы, дадим им условные названия и приведем примеры соответствующих ЯП.

Выделяют следующие факторы:

Уровень ЯП – обычно характеризует степень близости ЯП к архитектуре компьютера. Так, автокод (ассемблер) относят к ЯП низкого уровня; Фортран, Паскаль, Аду называют ЯП высокого уровня; Язык Сетл [44], созданный известным математиком Дж. Шварцем, служит примером ЯП «очень высокого уровня» (иногда говорят сверхвысокого уровня) – его базис составляют теоретико-множественные операции, далекие от традиционной архитектуры компьютеров. Встречаются и другие толкования уровня ЯП – это довольно расплывчатое, однако часто используемое понятие.

В «Науке о программах» Холстеда [45] сделана интересная попытка придать этому понятию точный смысл. Уровень ЯП, по Холстеду, определяется отличием программы на этом ЯП от простого вызова процедуры, решающей поставленную задачу. Выводится формула, численно выражающая уровень ЯП. Ясно, что в такой трактовке уровень ЯП непосредственно связан с классом решаемых задач – один и тот же ЯП для разных классов задач имеет разный уровень (что в целом согласуется с интуитивным понятием об уровне ЯП).

Специализация ЯП характеризует потенциальную или реальную область его применения. Различают ЯП общего назначения (или универсальные) и ЯП с более определенной специализацией. Классическими примерами универсального ЯП могут служить язык ассемблера ЕС или ПЛ/1. В свое время на эту роль претендовали Алгол-60, Симула-67, Алгол-68. Реально ее играют также Фортран,

в частности его стандарты – Фортран-66 (ГОСТ) и Фортран-77 (стандарт ИСО), Бейсик (в особенности его развитые модификации), Паскаль (в особенности диалекты, допускающие раздельную трансляцию), менее известные у нас такие ЯП, как Корал (стандарт МО Великобритании), Джовиал (стандарт ВВС США), а также отечественный Эль-76.

Более выраженную специализацию обычно приписывают таким ЯП, как Кобол (коммерческая); Рефал, Снобол, Лисп (символьная); Модула, Ада (реальное время). В Алголе-60 и Фортране также можно усмотреть специализацию (научные и инженерные расчеты).

Все названные ЯП в той или иной степени можно отнести к базовым ЯП широкого назначения. Обычно на их основе (или без них) строят более специализированные ПОЯ.

Алгоритмичность (процедурность) – характеризует возможность абстрагироваться от деталей (алгоритма) решения задачи. Другими словами, алгоритмичность тем выше, чем точнее приходится планировать выполняемые действия и их порядок (или синхронизацию); она тем ниже, чем более язык позволяет формулировать соотношения и цели, характеризующие ПО и решаемую задачу, оставляя поиск конкретного способа решения (способа достижения целей) за исполнителем.

Типичные примеры алгоритмического (процедурного) языка – ассемблер, Фортран, Ада; неалгоритмического (непроцедурного) языка – Пролог. Рефал занимает промежуточное положение – хотя мы рассматриваем его как естественное развитие марковских алгоритмов, многие воспринимают Рефал-предложения (особенно с мощными спецификаторами) как соотношения, удобные для непосредственного представления знаний о предметных областях, а поиск подходящего Рефал-предложения – как автоматический выбор подходящего способа решения задачи.

Динамизм (диалоговость, интерактивность) – характеризует степень изменчивости программных объектов в процессе выполнения программы. Частично мы обсуждали этот вопрос, когда занимались статическими, квазистатическими и динамическими характеристиками объектов. С этой точки зрения различаются статические, квазистатические и динамические ЯП.

Разделяют языки также по степени изменчивости текста программы. Один крайний случай – текст программы в процессе ее работы менять нельзя. Этот случай представлен «пакетными» языками (Фортран, Паскаль, Ада, Модула-2 и т. д.). Другой крайний случай – программист волен изменить программу на любом этапе ее исполнения. Этот случай представлен «диалоговыми» языками (Бейсик, Апл и более современными Визикалк, Лого и др.). Промежуточный вариант – программу в процессе ее исполнения может изменять лишь сама программа (но не программист). Это крайний случай пакетного динамизма (представлен языками Лисп, Инф и др.).

Связь концепции диалога с общей концепцией ЯП заслуживает дополнительного анализа. Суть – в том, что концепция ЯП как *средства планирования поведения исполнителя* в чистом виде не обязана предполагать диалога (создатель программы вполне может отсутствовать в период ее исполнения – и это типичный

для ЯП случай, тем более для ЯП индустриального программирования). Другими словами, концепция ЯП в общем случае не предполагает обратной связи исполнителя с создателем программы в процессе ее рабочего исполнения.

Концепция диалога обязана предполагать такую связь и поэтому в общем случае требует специфических выразительных средств, отличающихся от средств планирования (краткостью, менее жестким контролем, ориентацией на интерпретацию, использованием разнообразных органов чувств (слуха, зрения, осязания, двигательных навыков) и т. п.). Так что управление диалогом – ортогональный срез в системе средств общения с компьютером.

8.2.2. Недостатки традиционной классификации

Удовлетворительной классификации живых ЯП не существует. Тот или иной ярлык, присваиваемый ЯП в программистском фольклоре или литературе, в лучшем случае отражает лишь некоторые его характерные свойства. К тому же с развитием самого ЯП, сферы его применения, контингента пользователей, методов программирования, критериев качества программ и т. п. относительное положение ЯП, его оценка могут существенно измениться.

Например, Фортран начинался как язык высокого уровня для научно-технических расчетов. Однако его первый международный стандарт (ему соответствует отечественный ГОСТ 23056–78) выделяется уже не столько особой пригодностью для создания расчетных программ, сколько возможностью создавать мобильные (легко переносимые из одной среды в другую) программы практически произвольного назначения. Так что если хотите, чтобы ваша программа работала на любой машине и практически в любой операционной среде, пишите ее на стандарте Фортрана, руководствуясь правилами, изложенными, например, в [46].

Аналогична судьба и других классических языков. Их современная оценка зависит скорее не от технических характеристик, а от социальных (распространенность и качество реализаций, наличие устойчивого контингента пользователей в определенной области знаний, объем парка эксплуатируемых программ и т. п.).

Так, Бейсик и Паскаль, появившись как учебные ЯП с весьма ограниченной областью серьезных применений, стали, подобно Фортрану, практически универсальными ЯП на персональных компьютерах (еще одно подтверждение роли социальных факторов в судьбе ЯП – важно научить людей ими пользоваться, дальше действует принцип чайника).

8.2.3. Принцип инерции программной среды

К сожалению, системы программирования, поддерживающие разные ЯП, как правило, несовместимы между собой в том смысле, что нельзя написать часть программы на одном ЯП, часть – на другом или воспользоваться программой, написанной на другом ЯП. Если бы эта **проблема модульной совместимости** различ-

ных ЯП была решена, то только тогда технические характеристики ЯП приобрели бы существенный вес при выборе конкретного ЯП для решения конкретной задачи.

Сейчас же определяющим фактором при таком выборе служат не свойства задачи и ЯП как инструмента ее изолированного решения, а тот язык и та среда, с помощью которых обеспечены программные услуги, которыми необходимо пользоваться при решении задачи. Другими словами, действует **принцип инерции программной среды**: *развивать среду лучше всего ее «родными» средствами*. Еще одна модификация принципа чайника (или его следствие).

Так что при современном состоянии модульной совместимости выбор инструментального ЯП подчиняется принципу инерции среды и как самостоятельная проблема перед рядовым программистом обычно не стоит. К сожалению, многие учебники программирования не учитывают принципа инерции и по существу вводят читателей в заблуждение, уделяя излишнее внимание технической стороне проблемной ориентации ЯП.

8.2.4. Заповеди программиста

Сформулируем в краткой форме ответы на основные вопросы об оценке, выборе, использовании и создании ЯП.

1. Выбирай не столько базовый ЯП, сколько базовую операционную среду (с учетом потребностей всего жизненного цикла создаваемого изделия).
2. На основе выбранного базового ЯП создавай свой ПОЯ для каждой значимой задачи с учетом выбранной технологии.
3. ЯП тем лучше, чем дешевле с его помощью оказывать программные услуги.
4. Минимальное ядро ЯП плюс проблемно-ориентированные языковые модули – разумный компромисс сундука с чемоданчиком.

8.3. Тенденции развития ЯП

8.3.1. Перспективные абстракции

Переходя от классификации современных ЯП к тенденциям их развития, прежде всего отметим аналог принципа чайника: область ЯП в целом, с одной стороны, стремительно развивается, но, с другой – остается весьма консервативной. Первое касается в основном теоретических исследований и экспериментов в области языкотворчества, второе – практики массового программирования. Естественная инерция носителей традиционных ЯП, трудные проблемы совместимости и переноса программных изделий, недостоверность оценок выгоды от применения новых ЯП, высокая степень риска от неправильной оценки перспективности ЯП при многомиллионных затратах на комплексное освоение ЯП создают на пути широкого внедрения новых ЯП порог, преодолеть который за последние годы удалось лишь нескольким языкам (Си, Модула-2, Пролог, Фортран-77, Ада). Программисты в основном продолжают пользоваться традиционными языками (Бейсик, Паскаль, Фортран, Кобол, Лисп) и их диалектами, учитывающими новые воз-

можности аппаратуры (диалог, графику, параллелизм, цвет, звук) и новые языковые средства (развитую модульность, типизацию, наследование и др.).

Поэтому, говоря о тенденциях развития ЯП, уделим основное внимание достаточно отработанным идеям и концепциям, уже вошедшим в практику программирования, но, возможно, еще не завоевавшим всеобщего признания. С другой стороны, постараемся разделить тенденции, касающиеся свойств ЯП, непосредственно воспринимаемых программистом (внешних, технологических свойств ЯП), и тенденции, касающиеся внутренней проблематики ЯП, в значительной степени скрытой от программиста (внутренней, авторской).

Из внешних тенденций выделим *освоение перспективных абстракций*, а из внутренних – *стандартизацию ЯП*.

Среди других заслуживающих внимания тенденций отметим *освоение новых этапов жизненного цикла программных изделий*. С этой точки зрения характерен язык проектирования программ SDL/PLUS [15]. В нем самое для нас интересное – концепция непрерывного перехода от спецификации программы к ее реализации, оригинальное обобщение понятия конечного автомата, а также основанные на нем мощные средства структуризации взаимодействия процессов.

Подчеркнем, что ЯП в своем развитии отражают (с некоторым запаздыванием) квинтэссенцию современной философии программирования и в этом качестве воспринимают все его фундаментальные концепции.

Казалось бы, естественным развитием ЯП было бы освоение новых технических средств (графики, цвета, звука, манипуляторов, огромной памяти). Однако пока этот аспект не дал ничего особо интересного для ЯП. Возможно, сказывается отмеченное выше различие концепций ЯП и диалога, где названные средства стремительно осваиваются.

Некоторые тенденции развития ЯП может подсказать следующий перечень характерных примеров абстракции-конкретизации. В его первой колонке указано, от чего удается отвлечься с помощью средства абстракции, указанного во второй колонке, и средства конкретизации, указанного в третьей колонке. Сначала перечислены хорошо освоенные абстракции, затем – менее привычные, наконец – перспективные.

Аспект	Средство абстракции	Средство конкретизации
Освоенные абстракции		
а) размещение	имя	загрузчик, управление представлением
б) исполнение	процедура	вызов, специализатор
в) порождение	родовые объекты, макросы	настройка, макрогенератор макровызов
г) компьютер	ЯП (виртуальная машина)	транслятор (эмулятор)
Менее привычные абстракции		
д) контекст	пакет, модуль	указатель контекста
ж) реализация	спецификация	связывание (по именам)
з) представление	абстрактные ресурсы, типы данных (АТД)	спецификация представления

Аспект	Средство абстракции	Средство конкретизации
и) именование	образец, условие	ассоциативный поиск, конкретизация образа
к) исключения	нормальные сегменты	аварийные сегменты
л) взаимодействие процессов	последовательные сегменты	сигналы, семафоры, рандеву, каналы
м) ЯП	псевдокод	программист, конвертор
н) изменения программы	слой (по Фуксману)	слой изменений
Перспективные абстракции		
о) проблемная область	информатика	творец прикладной теории
п) информационный объект	теория, модель, система соотношений, база знаний	факты, база данных, дополнительные соотношения
р) задача	(вычислительная) модель	запрос, имена аргументов и результатов
с) программа	задача	значения аргументов

Поясним некоторые термины.

Абстракция от (прямого) именования (**«и»**) обеспечивается использованием *образцов* (вспомним модель МТ). Конкретизация (связь «косвенного имени» со значением) осуществляется *поиском* подходящего образца (если фиксировано значение) или подходящего значения (если фиксирован образец). Такой поиск называют **ассоциативным**. Он широко используется в современных ЯП [47, 48 и др.].

Конверторами (**«м»**) называют программы, переводящие с одного ЯП высокого уровня на другой. В частности, распространены конверторы с так называемых «структурных» расширений стандартных ЯП (Фортрана, ПЛ/1, Кобола). Они позволяют писать программу на псевдокоде, в значительной степени отвлекаясь от особенностей конкретного ЯП, а затем автоматически или полуавтоматически получать программу на стандартном ЯП.

Подход к абстракции от потенциальных изменений (**«н»**) программы (в процессе ее разработки) впервые сформулирован А. Л. Фуксманом в его концепции «расслоенного программирования» [33]. Программа строится как иерархия «слоев», каждый из которых реализует все более полный набор предоставляемых программой услуг. При этом последующие слои строятся как перечни изменений предыдущих слоев. Так что при создании (рассмотрении, изучении) очередного слоя удается абстрагироваться от последующих изменений программы, реализующих более развитые услуги (функции). Особенно важно, что каждый очередной слой работоспособен без последующих. Концепция расслоенного программирования поддержана ЯП АКТ [33]. Близка к ней по замыслу (и эффекту) и так называемая **инкрементная компиляция** (в сочетании с развитыми средствами поддержки проектов), реализованная в известной системе R1000 (хотя основная исходная мотивировка инкрементной компиляции – экономия перекомпиляций, а не рациональная структура программы). Концепцию расслоенного программи-

рования полезно сопоставить с современной концепцией наследования в ЯП, наиболее полно воплощенной в объектно-ориентированных ЯП.

Перспективные абстракции – вариация на тему одного из выступлений С. С. Лаврова.

Творец (прикладной) теории конкретной проблемной области **(«о»)** формулирует ее на некотором языке представления знаний. Например, знание о синтаксисе языка Ада фиксирует правилами БНФ. Это знание состоит из фактов, касающихся единичных объектов, и общих правил (соотношений), касающихся целых классов объектов. Примеры фактов: А – буква, 7 – цифра. Примеры соотношений: присваивание = левая_часть правая_часть. Работа творца теории существенно неформальная, поэтому разумно говорить лишь о частичной ее автоматизации.

Однако если теория записана на подходящем языке представления знаний, открываются богатые возможности для автоматизации рутинных этапов дальнейшей работы.

Во-первых, можно добавить факты и соотношения (**«п»**), характеризующие конкретный объект, соответствующий теории (в логике такой объект называется **моделью**, в технике – **конструкцией**). После этого полное описание объекта можно в принципе получать автоматически (конечно, при определенных ограничениях на теории и частичные описания объектов). Например, если задана «теория» языка Ада в виде совокупности правил БНФ, то достаточно написать текст конкретной программы, чтобы параметрический синтаксический анализатор был в состоянии по этой теории и фактам (конкретной последовательности литер) построить дерево вывода этой программы в синтаксисе Алы – полное описание конкретного информационного объекта (модели, конструкции), удовлетворяющее теории и дополнительным условиям.

Это возможно потому, что язык БНФ **разрешим** в том смысле, что существует алгоритм построения дерева разбора для любого входного текста и любого синтаксиса, написанного на БНФ.

При этом деревьев разбора может быть несколько и даже бесконечно много (или ни одного для неправильных текстов). Аналогично можно рассматривать систему уравнений в частных производных как теорию, краевые условия – как дополнительные факты и соотношения, решение – как модель теории, а соответствующий сеточный метод – как разрешающий алгоритм рассматриваемой теории. Однако здесь, как известно, не существует общего разрешающего алгоритма, применимого к любым уравнениям в частных производных при любых краевых условиях.

Когда модель построена (**«р»**), бывает интересно узнать ее свойства (обычно неизвестные до построения). Другими словами, на модели приходится решать конкретные классы задач. В случае дерева разбора такие классы задач возникают, например, при контекстном анализе программы и синтезе объектного кода (требуется найти объявления для выбранных идентификаторов программы, проверить соответствие числа формальных и фактических параметров при вызове процедуры и т. п.). Задачи этих классов можно при определенных условиях решать автоматически, не прибегая к составлению программ для решения каждой задачи (и даже не строя модель целиком).

Один из способов (**«с»**) состоит в том, что еще при описании теории указываются отношения вычислимости между компонентами моделей. Они говорят о том, что одни компоненты могут быть непосредственно вычислены по другим (с помощью процедур, явно заданных на обычном ЯП, например на Фортране или Паскале). Если теперь рассматривать только задачи вычисления компонент модели, то (при определенных ограничениях) появляется возможность автоматически подбирать «расчетные цепочки» из элементарных процедур, решающие поставленную задачу при любых возможных аргументах. Таким образом, появляется возможность настраиваться в конкретной модели на конкретный класс однородных задач, оставляя пока неопределенными значения аргументов (выделяющих конкретную задачу из этого класса).

В сущности, таким планировщиком «расчетных цепочек» служит и синтаксически управляемый компилятор, когда составляет объектную программу по дереву вывода из «элементарных» заготовок, поставляемых семантическими процедурами-трансдукторами. Но при этом решается очень узкий класс задач – создание исполнимой программы, аргументами которой служат затем ее входные данные.

8.3.2. Абстракция от программы (в концептуальном и реляционном программировании)

Классический пример **концептуального программирования** – решение треугольников. Теорией служат соотношения вычислимости между атрибутами треугольников (сторонами, углами, периметром, площадью) – им соответствуют явные процедуры; моделью – треугольник с соответствующими атрибутами; задачей – запрос на вычисление, например, площади по заданным сторонам (при этом указываются не значения сторон, а только их имена). Планировщик готовит расчетную цепочку. Остается последняя конкретизация – можно задавать значения сторон и получать соответствующие значения площади после выполнения расчетной цепочки.

Указанный подход реализован в отечественных системах ПРИЗ и СПОРА (ЯП, соответственно, Утопист и Декарт) [49, 50].

В инструментальном режиме работы этих систем программисты-конструкторы строят модель-конструкцию (выписывают необходимые соотношения вычислимости и соответствующие им процедуры), обеспечивая конечным пользователям возможность работать в функциональном режиме в рамках созданной модели-конструкции без обычного программирования. Так что в функциональном режиме пользователь определяет на модели задачу, указывая имена ее аргументов и результатов, а затем после работы планировщика эксплуатирует полученную расчетную цепочку.

Языки Утопист и Декарт обеспечивают абстракцию от программы и от задачи, но не от модели. Модель (точнее, ее процедуры) приходится явно программировать.

Другой подход предлагает реляционное программирование, о котором шла речь в **разд. 4.** Наиболее известный язык этого класса – Пролог. К этому же клас-

су относится отечественный Реляп. В них не требуется задавать элементарные процедуры и в общем случае не используется планировщик. Теория и «краевые условия» непосредственно используются для построения модели и ответа на запрос-задачу. Таким образом обеспечивается абстракция и от модели, и от задачи, и от программы. С этой точки зрения реляционное программирование более высокого уровня, чем концептуальное.

В самом начале книги в качестве одного из источников сложности программирования был указан семантический разрыв между ПО и языками компьютеров, из-за чего невозможно управлять компьютерами посредством целей, а не действий. Другими словами, «мир» обычного компьютера не содержит знаний о ПО (ее теории). Именно это обстоятельство не позволяет указывать цели и контролировать их достижение.

Концептуальное программирование в рамках конкретной модели позволяет обеспечивать «взаимопонимание» с компьютером (управлять его поведением) на уровне целей (постановок задач) и тем самым оказывается естественным шагом вперед по сравнению с традиционными ЯП с точки зрения как абстракции-конкретизации, так и прогнозирования-контроля. Подчеркнем, что достигается это за счет представления в компьютере знаний о проблемной области – фрагмента внешнего мира – в виде (разрешимой за счет планировщика и заготовленных процедур) модели-конструкции. При этом принципиальным с точки зрения предоставляемого уровня взаимопонимания оказывается само наличие в компьютере знаний о ПО (фрагментов ее теории), а выбранный уровень абстракции и необходимость явно программировать модель определяются в основном реализационными соображениями (стремлением к снижению ресурсоемкости программ).

Реляционное программирование «чище» в идейном отношении. Его ключевой принцип – разрешимость на уровне теории. Иначе говоря, после представления теории в компьютере – никакого программирования! Задачи решаются автоматически единым разрешающим алгоритмом. Проблемами ресурсоемкости предлагается заниматься «по мере их возникновения». В тех случаях, когда это важно, критичные модули реляционных программ можно переписать на традиционном ЯП. Даже если это неприемлемо, реляционная программа оказывается исключительно полезной как «исполняемая спецификация» традиционной реализации. Как уже отмечено выше, концептуальное, реляционное и объектно-ориентированное программирование удачно соединены в языке НУТ [38].

С учетом общей тенденции к освоению перспективных абстракций можно высказать следующий тезис: если на протяжении двух последних десятилетий в области ЯП основной была абстракция от компьютера, то в ближайшей перспективе основной станет абстракция от программы (ее можно назвать и абстракцией от реализации).

В этом смысле реляционные языки несколько неестественно называть языками программирования. Скорее, это разрешимые языки представления знаний. Однако существует традиция называть программами любое представленное в компьютерах знание (в том числе и теории, а не только алгоритмы).

8.3.3. Социальный аспект ЯП

Очевидно, что далеко не все абстракции со стр. 339 обеспечены соответствующими языковыми конструктами (и, по-видимому, некоторые никогда не будут обеспечены), но приведенный спектр абстракций дает возможность анализировать конкретный ЯП на предмет развития в нем аппарата определенных абстракций и тем самым судить о ЯП существенно более содержательно. Подчеркнем, что принцип технологичности требует не наивысшего, а оптимального уровня абстракции в соответствии с требованиями к ЯП.

Приведенный на стр. 339 перечень абстракций показывает важность социального аспекта ЯП. Например, абстрагироваться от компьютера – дело творцов ЯП, а вот средства конкретизации обеспечивают реализаторы трансляторов, авторы учебников и т. п. Другими словами, оценка разработанности аппарата абстракции-конкретизации в ЯП выходит за рамки его внутренних свойств, причем это может касаться важнейших для пользователя абстракций.

Свойства ЯП как социального явления (точнее, артефакта) подчеркивает также уже отмеченная изменчивость оценки ЯП, связанная с внешней по отношению к нему человеческой деятельностью – поддержкой, пропагандой, накоплением реализаций и программ, появлением хороших учебников, развитием возможностей аппаратуры и методов реализации, технологии программирования.

Упомянутую абстракцию от программы (от ее реализации, а не спецификации) полезно трактовать не только как абстракцию от программы для ее конечного пользователя, но и как абстракцию от необходимости (и возможности) знать программу для желающего ее развить.

Именно с последней трактовкой связан принцип, который можно назвать **принципом защиты авторского права**, – *ЯП должен способствовать защите авторских интересов создателей программных изделий и, в частности, гарантии качества предоставляемых услуг.* Этот принцип мы отмечали еще в связи с Адой, но свое почти идеальное воплощение он нашел в объектно-ориентированном программировании.

8.3.4. Стандартизация ЯП

Если освоение перспективных абстракций отражает стремление творцов ЯП предоставить программистам как можно более адекватные средства создания программ, то **стандартизация ЯП** нацелена прежде всего на расширение сферы применимости уже созданных программ, уже накопленных знаний и навыков, а также на создание определенных гарантий для работоспособности программ и сохранения квалификации программистов при изменении программной среды.

Таким образом, в идеале указанные две тенденции взаимно дополняют и уравновешивают друг друга. Если первую можно считать основным источником здорового *радикализма*, то вторую – основным источником здорового *консерватизма*. Важно понимать, что ни одну из них не следует оценивать изолированно, каждая вносит свой вклад в современное развитие ЯП.

Для творческих натур, какими обычно бывают программисты, часто понятнее и эмоционально ближе первая из отмеченных тенденций. Вторая для своей адекватной оценки требует не только большего психического напряжения, но и несравненно более высокой квалификации (и даже жизненного опыта). Поэтому, и по ряду других причин, стандартизация в области ЯП пока значительно отстает от потребностей практики как в мире, так и особенно в России. С другой стороны, в настоящее время это область с нетривиальной научной проблематикой, специфическими методами и техническими средствами, широким международным сотрудничеством и вполне осязаемыми достижениями. Среди последних – международные стандарты Фортрана, Паскаля, Ады, проекты стандартов Си, Бейсика, перспективного Фортрана, расширенного Паскаля и др. [51].

В среде программистов со стандартизацией ЯП связаны недоразумения, касающиеся всех ее аспектов – от целей до проблематики, применяемых методов и современного состояния дел. Посильный вклад в их устранение представлен в [30–32].

Заключение

Подведем итоги. *Во-первых*, мы смотрели на ЯП с нескольких различных позиций, стремясь к тому, чтобы взаимодействие этих позиций было продуктивным. Так, технологическая позиция постоянно давала материал для формулировки принципов и концепций, интересных прежде всего с позиции авторской. Таковы принцип цельности; принцип РОРИУС; концепция уникальности типа; понятие критичной потребности и неформальной теоремы о существовании ее решения; концепция регламентированного доступа (инкапсуляция); принцип реальности абстракций; принцип целостности объектов; концепция внутренней дисциплины доступа к разделяемым ресурсам; концепции единой модели числовых (и временных) расчетов; принцип защиты авторского права; концепция раздельной трансляции; динамический принцип выбора реакции на исключение; принцип динамической ловушки, концепция наследуемости, критерий ЕГМ и др.

Аналогичным образом семиотическая позиция взаимодействовала с авторской и технологической. А именно, занимаясь моделями Н, МТ, и Б, мы рассмотрели различные виды семантик. При этом дедуктивная семантика позволяет не только прояснить такой технологический элемент, как доказательство корректности программ, но и обосновать требования к управляющим структурам в ЯП. **Эти требования иногда неудачно называют принципами структурного программирования; такая узкая трактовка отвлекает внимание от корректной структуризации всех аспектов программирования, в частности структуризации данных.**

Во-вторых, имея дело со всеми моделями, мы, с одной стороны, старались демонстрировать возможность строить (выделять) достаточно четко фиксированные модели, критерии, оценки и способы рассуждений (в том числе убедительных обоснований, вплоть до строгого математического доказательства содержательных свойств моделей).

Но, с другой стороны, мы постоянно подчеркивали сложность ЯП как объекта конструирования и исследования, старались показать, как выводы о свойствах проектных решений зависят от точки зрения, от самых общих подходов к проектированию ЯП. Особенно наглядно это проявилось при сопоставлении принципов сундука и чемоданчика. Ведь оказались под сомнением такие ранее «обоснованные» решения, как указатель контекста, приватные типы и концепция рандеву, а затем даже перечисляемые типы.

ЯП как сложное явление реального мира (лингвистическое, техническое, социальное, математическое) всегда уязвимо *с точки зрения односторонней критики*. ЯП всегда – *плод компромиссов* между технологическими потребностями и реализационными возможностями. Продуктивное творчество в области ЯП – скорее высокое искусство, чем предмет точной инженерной или тем более математической науки. С другой стороны, возникнув как артефакты, творения отдельных людей или относительно небольших авторских коллективов, ЯП продолжают жить по законам, весьма напоминающим законы развития естественных языков.

Список литературы

1. Guidelines for the preparation of programming language standards // ISO/TC97/SC22 WG10. – 1986. – № 251. – July.
2. Joung J. An Introduction to ADA. – Ellis Horwood Ltd, 1983. – 256 p.
3. Ершов А. П. Трансформационная машина: тема и вариации // Проблемы теоретического и системного программирования. – Новосибирск, 1982. – С. 5–24.
4. Романенко С. А. Генератор компиляторов, порожденный самоприменением специализатора, может иметь ясную и естественную структуру. – М., 1987. – 35 с. (ИПМ им. М. В. Келдыша АН СССР, № 26.)
5. Turchin V. F. The concept of a supercompiler // ACM Transactions on Programming Languages and Systems. – 1986. – Vol. 8. – № 3. – P. 292–325.
6. Хьюз Дж., Мичтом Дж. Структурный подход к программированию / пер. с англ., под ред. В. Ш. Кауфмана. – М.: Мир, 1980. – 278 с.
7. Темов В. Л. Язык и система программирования Том. – М.: Финансы и статистика, 1988. – 240 с.
8. Замулин А. В. Язык программирования Атлант (предварительное сообщение). – Новосибирск, 1986. – 46 с. (ВЦ СО АН СССР, № 654.)
9. Замулин А. В. Типы данных в языках программирования и базах данных. – Новосибирск: Наука, 1987. – 150 с.
10. Клещев А. С., Темов В. Л. Язык программирования Инф и его реализация. – Л.: Наука, 1973. – 150 с.
11. Пентковский В. М. Автокод Эльбрус Эль-76. Принципы построения языка и руководство к пользованию / под ред. А. П. Ершова. – М.: Наука, 1982. – 350 с.
12. Йодан Э. Структурное проектирование и конструирование программ / пер. с англ., под ред. Л. Н. Королева. – М.: Мир, 1979. – 416 с.
13. Communications of ACM. – 1986. – Vol. 27. – № 12.
14. Янг С. Алгоритмические языки реального времени. Конструирование и разработка / пер. с англ., под ред. В. В. Мартынюка. – М.: Мир, 1985. – 400 с.
15. Язык спецификаций SDL/PLUS и методика его использования / Я. М. Барздинь, А. А. Калниньш, Ю. Ф. Стродс, В. А. Сыцко. – Рига: ВЦ ЛГУ им. Стучки, 1986. – 204 с. (Материал информационного фонда РФАП Латвии № ИН0047.)
16. Пайл Я. Ада – язык встроенных систем / пер. с англ., под ред. А. А. Красилова. – М.: Финансы и статистика, 1984. – 238 с.
17. Вегнер П. Программирование на языке Ада / пер. с англ., под ред. В. Ш. Кауфмана. – М.: Мир, 1983. – 240 с.
18. The Programming language Ada Reference Manual. American National Standards Institute, Inc. ANSI/MIL-STD-1815A-1983 // Lecture Notes in Computer Science. – 1983. – Vol. 155.
19. Вирт Н. Алгоритмы + структуры данных = программы / пер. с англ., под ред. Д. Б. Подшивалова. – М.: Мир, 1985. – 406 с.

20. Wirth N. Design a System from Scratch // Structured Programming. – 1989. – Vol. 1. – P. 10–18.
21. Wirth N. From Modula to Oberon // ETH-ZENTRUM, SWITZELAND. – 1988. – Tuesday 23 February. – P. 1–9.
22. Backus J. Can Programming Be Liberated from von Neumann Style? A Functional Style and Its Algebra of Programs // CACM. – 1978. – Vol. 21. – № 8. – P. 613–641.
23. Грис Д. Наука программирования / пер. с англ., под ред. А. П. Ершова. – М.: Мир, 1984. – 416 с.
24. Дейкстра Э. Дисциплина программирования / пер. с англ., под ред. Э. З. Любимского. – М.: Мир, 1978. – 275 с.
25. Клещев А. С. Реляционный язык как программное средство для искусственного интеллекта. – Владивосток, 1980. – 17 с. (НАПУ ДВНЦ АН СССР, № 26.)
26. Клоксин У., Меллиш К. Программирование на языке Пролог. – М.: Мир. 1987. – 336 с.
27. Попов Э. В. Экспертные системы. – М.: Наука, 1987. – 285 с.
28. Клещев А. С. Реализация экспертных систем на основе декларативных моделей представления знаний. – Владивосток, 1988. – 46 с. (ДВО АН СССР.)
29. Игнатьев М. Б., Потемкина А. А., Филоганов В. В. Параллельные алгоритмы и средства программирования: тексты лекций. – Л.: ЛИАП, 1987 – 50 с.
30. Hill L. D., Meek B. L. (eds) Programming Language Standardization. – Ellis Horwood Ltd, 1980. – 261 p.
31. Стандартизация языков программирования / А. Л. Александров, Л. П. Бабенко, В. Ш. Кауфман, Е. Л. Ющенко. – Киев: Техніка, 1989. – 160 с.
32. Кауфман В. Ш. Принципы стандартизации языков программирования // Программирование. – 1988. – № 3. – С. 13–22.
33. Фуксман А. Л. Технологические аспекты создания программных систем. – М.: Статистика, 1979. – 184 с.
34. Левин В. А. Проект базового языка спецификации Атон. – М., 1989. – 28 с. (ИПМ им. М. В. Келдыша АН СССР, № 117.)
35. Кауфман В. Ш., Левин В. А. Естественный подход к проблеме описания контекстных условий // Вестник МГУ. Серия «Вычислительная математика и кибернетика». – 1977. – № 2. – С. 67–77.
36. Дал У. И., Мюрхауг В., Нюгорд К. Симула-67. Универсальный язык программирования. – М.: Мир, 1969. – 99 с.
37. Андрианов А. Н., Бычков С. П., Хорошилов А. И. Программирование на языке Симула-67. – М.: Наука, 1985. – 288 с.
38. Tyugu E. H., Matskin M. B., Penjam J. E., Eomois P. V. NUT – an Object-Oriented Language // Computer and Artifical Intelligence. – 1986. – V. 5. – № 2. – P. 521–542.
39. Пратт Т. Языки программирования. Разработка и реализация / пер. с англ., под ред. Ю. М. Баяковского. – М.: Мир, 1979. – 575 с.
40. Брукс Ф. П. мл. Как проектируются и создаются программные комплексы / пер. с англ., под ред. А. П. Ершова. – М.: Наука, 1979. – 151 с.
41. Касьянов В. Н., Поттосин И. В. Методы построения трансляторов. – Новосибирск: Наука, 1986. – 344 с.

42. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов / пер. с англ., под ред. В. Н. Агафонова. – М.: Мир, 1979. – 656 с.
43. Кауфман В. Ш. О технологии создания трансляторов (проекционный подход) // Программирование. – 1980. – № 5. – С. 36–44.
44. Левин Д. Я. Сетл – язык весьма высокого уровня // Программирование. – 1976. – № 5. – С. 3–9.
45. Холстед М. Х. Начала науки о программах / пер. с англ., под ред. В. М. Юфы. – М.: Финансы и статистика, 1981. – 128 с.
46. Горелик А. М., Ушкова В. Л., Шура-Бура М. Р. Мобильность программ на Фортране. – М.: Финансы и статистика, 1984. – 167 с.
47. Грисуолд Р., Поудж Дж., Полонски И. Язык программирования СНОБОЛ-4 / пер. с англ., под ред. Ю. М. Баяковского. – М.: Мир, 1980. – 268 с.
48. Пильщиков В. Н. Язык плэнер. – М.: Наука, 1983. – 208 с.
49. Тыугу Э. Х. Концептуальное программирование. – М.: Наука, 1984. – 256 с.
50. Бабаев И. О., Новиков Ф. А., Петрушина Т. И. Язык Декарт – входной язык системы СПОРА // Прикладная информатика. – М.: Финансы и статистика, 1981. – Вып 1. – С. 35–73.
51. ISO I539-80(E). Programming Languages – FORTRAN.
52. ISO 7185-83(E). Programming Languages – PASCAL.
53. ISO 8652-87. Programming Languages – Ada.
54. ISO DP 9899. Programming Languages – C.
55. ISO DP 10279. Programming Languages – Basic.
56. ISO DP 1539. Programming Languages – FORTRAN.
57. ISO DP 10206. Programming Languages – Extended PASCAL.

Полезная литература, на которую прямых ссылок в тексте нет

1. Лавров С. С. Основные понятия и конструкции языков программирования. – М.: Финансы и статистика, 1982. – 22 с.
2. Шрейдер Ю. А. Логика знаковых систем (элементы семиотики). – М.: Знание, 1974. – 128 с.
3. Сафонов В. О. Языки и методы программирования в системе Эльбрус / под ред. С. С. Лаврова. – М.: Наука, 1989. – 390 с.
4. Лисков Б., Гатэг Дж. Использование абстракций и спецификаций при разработке программ / пер. с англ. – М.: Мир, 1989. – 424 с.
5. Фуги К., Судзуки Н. Языки программирования и схемотехника // СБИС и Мир. – 1988. – 224 с.
6. Цаленко М. Ш. Моделирование семантики в базах данных. – М.: Наука, 1989. – 288 с.
7. Бар Р. Язык Ада в проектировании систем. – М.: Мир, 1988. – 320 с.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**; электронный адрес **books@alians-kniga.ru**.

Кауфман Виталий Шахнович

Языки программирования

Концепции и принципы

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 06.07.2010. Формат 70×100 $^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 43,5. Тираж 1000 экз.
№

Web-сайт издательства: www.dmk-press.ru