

БИБЛИОТЕЧКА
ПРОГРАММИСТА

Ю. М. БЕЗБОРОДОВ

Индивидуальная отладка программ



БИБЛИОТЕЧКА ПРОГРАММИСТА

Ю. М. БЕЗБОРОДОВ

ИНДИВИДУАЛЬНАЯ ОТЛАДКА ПРОГРАММ

Под редакцией А. П. ЕРШОВА



МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1982

22.18
Б 34
УДК 519.6

Индивидуальная отладка программ. Безбородов Ю. М.—
М.: Наука. Главная редакция физико-математической литературы,
1982.— 192 с.

Книга посвящена отладке программ средней сложности или небольших блоков программных систем. Излагаются методика и типовые средства выполнения основных работ, составляющих отладку программ, рассматриваются методы и приемы разработки программ, предупреждающие появление ошибок и облегчающие их обнаружение и устранение. Общая методика отладки, излагаемая в книге, как правило, не зависит от используемых алгоритмических языков и конкретных отладочных средств, но примеры даются для PL/1, фортрана, алгола и для ОС ЕС ЭВМ.

Илл. 17, Табл. 2, Библ. 16 назв.

Б 1502000000—018 49-82
053 (02)-82

© Издательство «Наука»,
Главная редакция
физико-математической
литературы, 1982

ОГЛАВЛЕНИЕ

От редактора	5
Предисловие	7
Введение	9
0.1. Этапы решения задачи на ЭВМ	9
0.2. Необходимость отладки	17
Глава 1. Контроль программы	22
1.1. Контроль текста	22
1.1.1. Просмотр (22). 1.1.2. Проверка (23). 1.1.3. Прокрутка (23). 1.1.4. Печать текста (29). 1.1.5. Трансляция (29). 1.1.6. Статический анализ (30).	
1.2. Контроль результатов	30
1.2.1. Тестирование (30). 1.2.2. Правила тестирования (32). 1.2.3. Типы тестов (37).	
1.3. Классификация методов контроля	38
Глава 2. Локализация ошибок	39
2.1. Средства локализации	41
2.1.1. Аварийная печать (41). 2.1.2. Печать в узлах (48). 2.1.3. Слежение (52). 2.1.4. Прокрутка (55). 2.1.5. Другие средства (57). 2.1.6. Классификация средств (57).	
2.2. Использование средств	61
2.2.1. Сравнение средств (61). 2.2.2. Идентификация печати (62). 2.2.3. Минимизация печати (62). 2.2.4. Автоматизация сверки (63). 2.2.5. Включение отладочных средств (65).	
2.3. Методика локализации	66
2.3.1. От симптома к ошибке (66). 2.3.2. Обратное отслеживание (67). 2.3.3. Природа ошибки (69). 2.3.4. Некоторые замечания (70).	
Глава 3. Исправление ошибок	72
3.1. Виды исправлений	72
3.1.1. Исправление алгоритма (72). 3.1.2. Исправление программы (73).	
3.2. Средства исправления программ	74
3.2.1. Построчные исправления (74). 3.2.2. Контекстные исправления (77).	
3.3. Классификация видов исправлений	79

Глава 4. Предупреждение ошибок	80
4.1. Традиционные приемы	80
4.1.1. Этапность разработки (80). 4.1.2. Уровень языка (81). 4.1.3. Наглядность текста программы (82). 4.1.4. Личные приемы (83). 4.1.5. Стандарты программирования (84). 4.1.6. Защита от ошибок (84).	
4.2. Модульность программы	86
4.2.1. Модульность (86). 4.2.2. Реализация модульности (86). 4.2.3. Строение программ (88). 4.2.4. Трудности (89). 4.2.5. Локальность исправлений (92). 4.2.6. Хранение модулей во внешней памяти (95).	
4.3. Структурированность программы	96
4.3.1. Стандартные структуры (96). 4.3.2. Простота программы (98). 4.3.3. Дополнительные структуры (99). 4.3.4. Трудности (100).	
Глава 5. Нисходящее проектирование	103
5.1. Общие положения	103
5.1.1. Сущность нисходящего проектирования (103). 5.1.2. Первый пример (104). 5.1.3. Очевидность решений (116). 5.1.4. Обобщенные данные (117).	
5.2. Характерные черты	118
5.2.1. Второй пример (118). 5.2.2. Модульность и структурированность (127). 5.2.3. Блок-схема и псевдокод (128).	
5.3. Особенности применения	129
5.3.1. Трудности и преимущества (129). 5.3.2. Нисходящее тестирование (131).	
Глава 6. Примеры разработки и отладки	134
6.1. Преобразование текста	134
6.1.1. Постановка задачи (134). 6.1.2. Проект (134). 6.1.3. Алгоритм (136). 6.1.4. Программа (139). 6.1.5. Отладка (144).	
6.2. Перевод программы	146
6.2.1. Техническое задание (146). 6.2.2. Проект (146). 6.2.3. Алгоритм (151).	
Приложение. Препроцессорные средства	159
П.1. Общие сведения (159). П.2. Пример (160). П.3. Основные элементы и выражения (162). П.4. Основные операторы (166). П.5. Процедура и обращение к ней (171). П.6. Оператор включения текста (174). П.7. Программа и задание (175). П.8. Синтаксические формы (181). П.9. Сравнение препроцессорных и процессорных средств PL/1 (185).	
Литература	188
Предметный указатель	189

ОТ РЕДАКТОРА

Если рассматривать программирование как деятельность, то сразу станет видно, что в этой деятельности отладка зачастую оказывается самым длительным и трудным этапом. Согласившись с этим наблюдением, приходится только удивляться тому, что в огромном потоке литературы по программированию публикации, специально посвященные отладке, занимают более чем скромное место. Представляется уместным немного порассуждать по этому поводу.

Во-первых, вообще, изучение программирования как деятельности сильно отстает от изучения программирования как совокупности алгоритмических, языковых и конструкторских средств решения задач на ЭВМ. Мы больше говорим о программных структурах как таковых, нежели о способах и путях их актуализации. Положение, надо сказать, начинает меняться в последние годы, и то, что называют методологией программирования, все в большей степени делает предметом изучения процесс программирования.

Во-вторых, «большая наука» программирования, которая в своем крайнем выражении отрицает эмпирический, рукотворный характер программного продукта, отрицает вместе с этим и отладку как последовательность экспериментов, направленных на получение достоверного знания о правильности программы. Как говорит Э. Дейкстра, «Отладка может лишь указать на наличие ошибок в программе, но никогда — на их отсутствие». Разрабатываются методы систематического извлечения программы из достоверного исходного знания о способе решения задачи, при этом таким способом, что программист не в состоянии внести неверную команду в программу, если отвлечься от синтаксических ошибок. Х. Миллз, описывая такую методику, дает следующее броское название своему докладу: «Как писать правильные программы и знать об этом».

В-третьих, слабый профессионализм большинства программистов затрудняет им преодоление наивного оптимизма в самооценке своего труда, при котором предусмотрительное ожидание ошибки постоянно заслоняется верой в близость конечной цели. Эта психологическая установка приводит к тому, что этап отладки воспринимается как нежелательный, чем-то невязанный извне, о котором следует поскорее забыть после окончания работы над программой.

Наконец, поиск общезначимых истин в отладке затрудняется тем, что, пожалуй, в способах предотвращения, обнаружения и исправления ошибок в наибольшей степени сказываются индивидуальные свойства программиста: его характер, темперамент, общий логический уровень и собственный опыт.

В свете сказанного с тем большим энтузиазмом следует приветствовать появление книги, посвященной индивидуальной отладке программ. Эта книга обладает двумя бесспорными достоинствами: она написана на основе реального и очень большого опыта; она рассчитана на операционную обстановку, в которой работает большинство программистов, — операционную систему Единой системы ЭВМ.

Хотелось бы отметить и литературные качества книги, не так часто проявляющие себя в современной торопливой научно-технической прозе. В изложении достигнут в целом убедительный баланс между справочной информацией, демонстрацией примеров и авторскими комментариями или вступительными замечаниями. Все это нанизано на очень четкую линию изложения, превращающую изучение технического материала в увлекательное чтение. Обращает на себя внимание определенность авторских рекомендаций. Опытному программисту, имеющему свой взгляд на отладку, эта определенность поможет яснее осознать его собственную систему; для начинающего читателя это качество книги послужит надежным путеводителем в поистине грандиозном конгломерате проблем, с которыми сталкивается программист, приступающий к серьезной работе в контексте ОС ЕС.

Книга состоит из двух практически равных частей. В первой части рассказывается, как искать, изолировать и исправлять ошибки. Во второй части описываются рабочие процедуры программирования, помогающие предотвращать ошибки. Автор занимает очень сдержанную позицию в вечном споре боевых и штабных офицеров — где решается судьба сражения: на поле боя или в штабе — другими словами, за пультом машины или за письменным столом. Очевидно, что каждый программист решает для себя эту дилемму индивидуально. Заслуга автора в том, что он, оставляя за машиной окончательное суждение о пригодности программы к работе, убедительно показывает продуктивность и экономность «домашнего анализа» и внимательного «просто чтения» программы.

Еще раз подчеркнем, что главная ценность его книги — это то, что она основана на реальном опыте производственного программирования и опирается на технические средства, реально доступные в наших вычислительных центрах. С этими средствами нам предстоит работать немало лет, и книга, которую читатель держит в руках, окажет ему в этой работе немалую помощь.

Академгородок
Август 1981 г.

А. П. Еришов

ПРЕДИСЛОВИЕ

В последнее время в связи с созданием больших программных систем возрос интерес к методике разработки и, в частности, отладки программ, о чем свидетельствует и появление ряда работ [1—3], посвященных этой теме. Было описано несколько подходов к организации отладки и предложен ряд общих методов разработки, способствующих упрощению отладки и успешному ее завершению [4—9].

Но методика разработки и отладки программных систем должна дополняться, конечно, и методикой изготовления и отладки отдельных программных блоков, подпрограмм, модулей, разрабатываемых одним программистом. Без применения эффективных способов создания таких программных единиц нельзя надеяться успешно решить и проблему создания программных комплексов.

Проблема отладки существует (существовала и ранее) также для программ средней сложности, которые в большинстве своем разрабатываются в научно-исследовательских и проектных институтах различного профиля. Конечно, для таких программ эффективность и достоверность отладки не является столь жизненно необходимой, и обнаружение серьезных ошибок в ходе эксплуатации программы не приводит к столь печальным последствиям, как для больших систем, так как автор программы обычно бывает в состоянии исправить их в приемлемые сроки.

Таким образом, вопросы повышения надежности программ, ускорения их отладки и разработки являются по-прежнему актуальными как для профессиональных программистов, работающих над отдельными блоками программных систем, так и для научных работников и инженеров, самостоятельно разрабатывающих свои программы.

Именно на таких читателей рассчитана эта книга, именно на такой подход к отладке указывает термин «индивидуальная» в названии книги. Вопросы разбиения программ на независимые части, особенности комплексной отладки затрагиваются лишь по отношению к работе одного программиста.

Поскольку эффективность отладки самым существенным образом зависит от общей методики разработки программ, в книге рассматриваются также некоторые современные методы проектирования программ, направленные на предупреждение и облегчение обнаружения ошибок на различных этапах разработки.

Книга состоит из введения, 6 глав и приложения. Введение касается, в основном, вопросов взаимосвязи этапа отладки с другими этапами. В главах 1—3 излагаются методика и средства выполнения основных работ, составляющих этап отладки: контроль программы, локализация ошибок и их исправление. В главах 4 и 5 кратко рассматриваются такие способы программирования, которые способствуют предупреждению появления ошибок в программе и быстрейшему их обнаружению: читатель познакомится с идеями модульного, структурного и, более подробно, нисходящего программирования. В главе 6 даются примеры разработки программ с применением изложенных в книге методов. Приложение посвящено изложению препроцессорных операторов, которые широко применяются в современных методах изготовления и отладки программ и часто упоминаются в книге.

Основные вопросы методики отладки излагаются, как правило, независимо от конкретных алгоритмических языков, операционных систем и ЭВМ, но примеры даются для PL/1, фортрана-IV, алгола-60 и для ОС ЕС ЭВМ [10—15].

Книга имеет целью помочь читателям в практической работе при решении различных задач на ЭВМ и привлечь их внимание к методическим вопросам разработки и, в особенности, отладки программ.

Автор выражает глубокую благодарность Н. П. Трифонову, В. В. Островскому, А. Х.-М. Алдакову, Т. Г. Лаврентьевой, В. И. Черезовой, чьи советы, критика и замечания значительно способствовали улучшению содержания книги. Автор признателен также научному редактору О. Ю. Меркадер и всем, кто оказывал помощь в работе над книгой.

Ю. М. Безбородов

*Человеку свойственно ошибаться,
а глупцу настаивать на ошибке.
Из античной мудрости (Ц и ц е р о н)*

*В каждой программе есть по крайней
мере одна ошибка.
Из фольклора программистов*

ВВЕДЕНИЕ

При решении задачи с использованием ЭВМ под отладкой программ понимается обычно один из этапов решения, во время которого с помощью машины происходит обнаружение и исправление ошибок, имеющих в программе; в ходе отладки программист убеждается в том, что его программа работает правильно.

Перед тем как приступить к изложению вопросов, связанных с отладкой, рассмотрим основные этапы решения задачи с применением ЭВМ и взаимодействие этих этапов между собой. Особое внимание при этом уделим связи различных этапов с этапом отладки и тем работам программиста, которые касаются предупреждения и устранения ошибок в ходе разработки программы.

0.1. Этапы решения задачи на ЭВМ

0°. *Постановка задачи.* Задача, которую предстоит решить программисту на ЭВМ, формулируется им самим или выдается ему в виде специального задания на разработку программы. Задание содержит формулировку задачи, необходимые характеристики разрабатываемой программы, требования к взаимодействию с ней. Выдаче такого задания для крупных задач может предшествовать большая работа научно-исследовательского характера.

Задание на разработку программы по форме и характеру должно быть аналогично техническому заданию (ТЗ) на разработку какого-либо технического продукта (см., например, ГОСТ 19.201-78 Единой системы программной документации).

Техническое задание полезно и в том случае, когда заказчик и исполнитель работают в одной и той же комнате или даже являются одним и тем же лицом. Наличие четкой письменной формулировки будет препятствовать подмене или отходу в процессе разработки программы от сформулированных в ТЗ требований в угоду

каким-то другим побочным целям. Кроме того, письменно сформулированное задание делает возможным обсуждение, оценку или согласование с заказчиками (пользователями) корректировку отдельных требований ТЗ в ходе разработки программы. ТЗ препятствует проникновению в программу таких ошибок и противоречий, которые могут быть обнаружены только после разработки большей части программы или уже на стадии анализа полученных результатов счета. Чем более формализованным по характеру будет техническое задание, тем больше шансов, что разрабатываемая программа будет решать именно ту задачу, которую имел в виду заказчик.

Техническое задание должно содержать также требования или указания, касающиеся принципов проверки и испытаний готовой программы.

1°. *Составление проекта.* На основании анализа технического задания программист выбирает основной метод решения задачи, составляет общий проект программы. Выбранный подход к решению задачи должен обеспечивать правильные результаты для тех условий функционирования программы, которые определены ТЗ, гарантировать требуемую скорость работы, предусматривать удобство использования программы и т. п.

В проекте, помимо формулировки выбранного общего метода решения задачи, характеризуются основные части проектируемой программы, их функции, взаимосвязь и последовательность выполнения, а также точно определяются входные данные и выдаваемые результаты как всей программы, так и основных ее частей. Поскольку каждая разрабатываемая программа, как правило, используется в дальнейшем не только ее автором, но и другими программистами, составляется и проект инструкции для пользователей, в которой фиксируется (и, таким образом, может быть заранее оценен и исправлен) предполагаемый режим общения пользователя (и оператора) с программой.

Для очень простых задач проектирование может производиться программистом мысленно, без фиксации на бумаге. Наоборот, очень сложные задачи могут потребовать нескольких этапов проектирования: предэскизное, эскизное, техническое — по аналогии с инженерным проектированием.

З а м е ч а н и е. Этапы 0° и 1° для несложных задач мало зависят от ЭВМ и ОС, с помощью которых предполагается решать задачу, а также от языка, на котором она будет программироваться; выбор ЭВМ и языка может быть произведен и на следующих этапах разработки.

2°. *Алгоритмизация.* На этот этап иногда смотрят как на вспомогательный, подготовительный к выполнению следующего этапа, считающегося основным (см. 3°), на котором производится написание

программы на выбранном языке программирования. Введение такого «промежуточного» этапа преследует цель облегчить выполнение этапа 3° и тем самым предотвратить возникновение многих ошибок при его осуществлении для сложных задач. Формулировка алгоритма закрепляет последовательность основных шагов выполнения программы, четко фиксирует функциональное содержание ее частей, позволяет уделить необходимое внимание простоте логической структуры разрабатываемой программы. Этап алгоритмизации является совершенно необходимым также в случае, если язык, на котором предстоит программировать, не вполне освоен программистом-разработчиком и предвидятся трудности при его использовании на следующем этапе (3°).

При разработке алгоритма необходимо учитывать ресурсы используемой ЭВМ (ее скорость, память) и возможности применяемой для решения задачи операционной системы. Алгоритмы для несложных задач, требования которых к ресурсам невелики, являются обычно машинно-независимыми.

В ходе разработки общего алгоритма используется некоторый специальный язык, который по своему характеру является промежуточным, переходным между неформальным, словесным способом изложения метода решения задачи на этапе 1° и формальным алгоритмическим языком для программирования на этапе 3°. Промежуточный язык должен сочетать в себе, с одной стороны, наглядность для отображения содержания и смысла выполняемых в алгоритме действий («что делается») и, с другой стороны, формализм для указания конкретных операций и последовательности их выполнения («как делается»).

В качестве такого промежуточного языка обычно используют блок-схемы, которые позволяют наиболее наглядно представить логическую структуру разрабатываемой программы, взаимосвязь отдельных частей программы, условия или кратность выполнения таких частей. Для отображения вычислительной (арифметической) стороны программы используются обычные математические средства или элементы алгоритмических языков, а в самых общих блок-схемах — просто словесная формулировка; иногда используются и все эти способы вместе.

Для достаточно сложных программ алгоритмизация проводится в несколько шагов с целью постепенной детализации алгоритма. Критерием окончания детализации при этом является то, что для каждого (псевдо)оператора в полученном на очередном шаге алгоритме программист уже имеет четкое и конкретное мысленное представление о том, как этот оператор алгоритма может быть выражен средствами выбранного языка программирования на этапе 3°. Для простых задач обычно разрабатывают блок-схемы на двух уровнях:

общая блок-схема программы и блок-схемы отдельных частей (блоков) программы.

После последнего шага детализации алгоритма (а иногда и после отдельных крупных шагов) проводится проверка полученного алгоритма для выявления допущенных ошибок. Методы контроля алгоритма аналогичны некоторым методам контроля программы и будут рассмотрены в главе 1.

В ходе разработки алгоритма, возможно, придется уточнять или изменять решения, принятые на этапе 1°, и в этом случае такие изменения обязательно вносятся в проект, который всегда должен соответствовать разрабатываемому алгоритму.

3°. Программирование. В случае, когда на предыдущем этапе был получен детально разработанный алгоритм, составление программы на выбранном для программирования языке (алгоритмическом языке высокого уровня, автокоде, языке ассемблера или машинном языке) сводится к п е р е в о д у этого алгоритма на язык программирования. Основные трудности и, следовательно, причины ошибок на этом этапе заключаются, во-первых, в необходимости знания всех требований и ограничений выбранного языка программирования и, во-вторых, в необходимости постоянного внимания ко многим деталям языка, которые приходится учитывать в ходе написания программы. Если этап 2° был выполнен некачественно и алгоритм представлен недостаточно детально, то его доводку придется выполнять «на ходу», во время программирования. Это затруднит процесс программирования-перевода и поведет к возникновению дополнительных ошибок в программе. Чем более процесс программирования будет походить на перевод, чем более механическим будет такой перевод, тем более легким будет составление программы и тем меньше возникнет ошибок на этом этапе, самом щедрым на ошибки. (Этап программирования, т. е. собственно процесс н а п и с а н и я программы, в последнее время все чаще называют *кодированием*, подчеркивая тем самым его механический, нетворческий характер).

После составления программы проводится ее проверка для обнаружения и исправления ошибок, внесенных на этом этапе. Если при проверке обнаруживаются ошибки, допущенные на предыдущем этапе (2°), то соответствующие исправления вносятся и в алгоритм, поскольку к нему еще придется обращаться на следующих этапах, и тексты алгоритма и программы должны соответствовать друг другу.

4°. Препарация. После составления программы производится ее перенос на машинные носители (перфокарты, перфоленту, магнитную ленту или диск), т. е. подготовка программы к выполнению ее на ЭВМ; будем называть [этот этап *препарацией*. Для предупреждения и сокращения ошибок препарации текст программы дол-

жен быть написан ясно и четко: чем небрежнее будет написан текст программы, тем больше ошибок возникнет в препарированной программе. Проверка правильности препарации осуществляется распечаткой программы, введенной в ЭВМ с использованных носителей, и последующей сверкой с исходным текстом.

Распечатка программы может производиться по специальной автономно (вне операционной системы) работающей на машине программе или с помощью одной из системных сервисных программ. Текст исходной программы можно получить и при последующей трансляции составленной программы (см. ниже 5°), но предварительная распечатка удобнее, поскольку при обнаружении грубой синтаксической ошибки транслятор может прекратить свою работу и печать текста или выдать чрезвычайно много диагностики, которая весьма затруднительна для разбора и является в большинстве своем бесполезной.

Как видим, рассматриваемый этап в отличие от предыдущих уже требует для своего осуществления непосредственного использования ЭВМ или, по крайней мере, ее внешних устройств.

5°. *Трансляция.* Транслятор в ходе осуществления трансляции, наряду с печатью транслируемой программы, производит поиск синтаксических ошибок в программе и, в случае их обнаружения, печатает диагностику, помогающую последующей локализации ошибок. Трансляция, а вместе с ней и поиск синтаксических ошибок, могут быть прекращены, если найдена очень грубая (с точки зрения транслятора) ошибка. Отсутствие синтаксических ошибок не говорит о том, что в программе нет ошибок препарации (например, вместо знака * был отперфорирован + или записана не та буква и т. п.). Поэтому тщательная сверка напечатанной программы с исходным текстом всегда необходима на данном или предыдущем этапе. Первые трансляции вновь составленной программы производятся обычно с включением таких режимов транслятора, которые позволяют получить текст программы вместе с дополнительной информацией о программе (например, с таблицей используемых в ней идентификаторов) для наиболее полной ее сверки.

В тех случаях, когда в распоряжении программиста имеется несколько трансляторов, сначала выбирается тот, который представляет больше возможностей для проведения начинающейся отладки. После окончания отладки, уже с помощью оптимизирующего транслятора производится перетрансляция с целью изготовления экземпляра программы, используемой в дальнейшем для счета (см., например, трансляторы уровней G и H для фортрана EC, а также уровней F и H для PL/1 EC).

6°. *Отладка.* На этапе отладки, которому, собственно, и посвящена эта книга, производится обнаружение с помощью ЭВМ ошибок

в программе и их исправление. Этап отладки можно разделить на 3 подэтапа:

6.1°. Контроль правильности программы.

6.2°. Локализация ошибок.

6.3°. Исправление ошибок.

На (под)этапе 6.1° — контроль программы — путем пропуска на машине специальных контрольных примеров устанавливается факт отсутствия или, в противном случае, наличия ошибок в программе. Здесь речь идет о содержательных (семантических) ошибках, которые не проявляются при трансляции программы.

На этапе 6.2° — локализация ошибок — точно устанавливается место, где в программе допущена ошибка (ошибки), последствия которой проявились при выполнении этапа 6.1°.

На этапе 6.3° производится исправление ошибок, выявленных на этапе 6.2°. Исправления вносятся как в программу, так и в алгоритм, если он затрагивается этими исправлениями.

Перечисленные подэтапы могут повторяться многократно (включая и этап трансляции, точнее перетрансляции), до тех пор пока контроль покажет, что ошибок в программе, по-видимому, нет.

В дальнейшем все эти подэтапы будут рассмотрены подробно (см. гл. 1, 2, 3).

З а м е ч а н и е. Поиск (и исправление) ошибок в программе происходит и на более ранних этапах ее разработки, но там он имеет подготовительный характер и отличается тем, что основным материалом при этом является текст программы, а не результаты ее работы.

7°. Оформление программы. Для возможности эксплуатации программы кем-либо кроме автора она должна быть оформлена: составлено ее описание, изготовлены машинные носители для передачи программы пользователям, разработана процедура дублирования носителей (см. ГОСТ 19.101-78 и др.). В описание включается инструкция по использованию программы, излагается примененный метод решения, приводятся алгоритмы (иногда и текст программы), а также контрольные примеры с эталонными результатами. Наличие описания программы позволяет не только успешно эксплуатировать ее длительное время, но и проводить ее модернизацию и использовать в дальнейших разработках. Основную часть описания составляют материалы, с которыми шла работа на предыдущих этапах разработки (проект разработки и описание метода решения, общая блок-схема, алгоритмы, проект инструкции для пользователя и т. п.). Поэтому для ускорения этапа оформления все перечисленные материалы всегда должны быть в рабочем состоянии и по содержанию вполне соответствовать друг другу и отлаживаемой программе; кроме того, уже на этапах разработки их нужно представ-

лять в таком виде, чтобы они могли быть использованы для описания программы без дополнительных переделок.

В случае, когда программа проста и предназначена для эксплуатации только ее автором, оформление программы может производиться уже после проведения счета по ней, одновременно с изготовлением отчета (см. ниже).

8°. *Счет.* По окончании отладки и оформления программы начинается ее эксплуатация: производится счет по ней, обычно многократный. Первые полученные результаты реальных расчетов подвергаются тщательному анализу, чтобы убедиться в пригодности использованного метода и установить согласованность полученных результатов с имеющимися данными и теорией.

Если правильность получаемых результатов не вызывает сомнений и эффективность программы удовлетворительна, то ее эксплуатация продолжается по мере необходимости. Но случается и так, что приходится снова рассматривать вопросы правильности разработанного алгоритма или пригодности реализованного метода, и тогда вся работа может вернуться к началу.

9°. *Отчет о работе.* На основании результатов, полученных в ходе эксплуатации программы, составляется отчет о проделанной работе, оценивается выбранный метод решения задачи и эффективность программы; публикуются научные выводы.

10°. *Модернизация.* Если разработчик программы постоянно работает в некоторой области науки или техники, то обычно рано или поздно наступает такой момент, когда перед ним возникает вопрос о модернизации старой программы или о составлении новой, развивающей идеи, реализованные в прежней программе. Модернизация программы проходит те же этапы, что и разработка, и начинается с составления технического задания на модернизацию. Успешное осуществление модернизации зависит от того, насколько легко можно будет при разработке новой программы использовать блоки старой программы и вносить в них изменения. Быстрое выполнение такого рода работ зависит, в свою очередь, как от структуры модернизируемой программы, так и от качества ее оформления (наличие описания программы, подробных алгоритмов, пояснений к программе и т. п.).

С х е м а р е ш е н и я з а д а ч и. На рис. 1 графически представлен процесс решения задачи на ЭВМ.

Стрелками показана взаимосвязь этапов: сплошные стрелки указывают на обычную, нормальную последовательность выполнения этапов, пунктирные — на связь этапов при обнаружении грубых ошибок в предыдущих этапах. Таким образом, нормальную последовательность выполнения этапов можно представить так:

{0°, 1°, 2°, 3°, 4°, {5°, 6.1°, 6.2°, 6.3°}, 7°, {8°}, 9°, 10°},

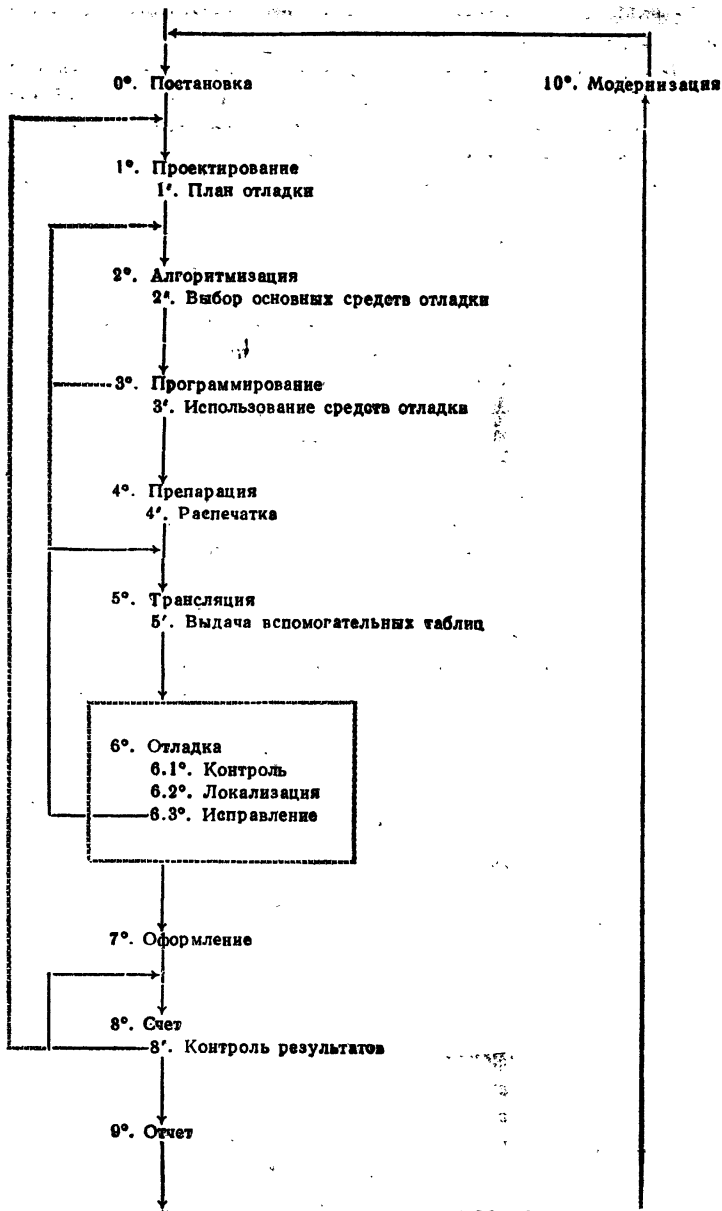


Рис. 1. Этапы решения задачи на ЭВМ.

где фигурные скобки указывают на циклическое повторение заключенных в них этапов.

Штрихи на рисунке отмечают работы внутри соответствующих этапов, связанные с поиском ошибок, допущенных на данном этапе или на предыдущих, а также работы, имеющие целью облегчить поиск ошибок в ходе предстоящей отладки (см. на рисунке подпункты 1', 2', 3', 4', 5', 8'). Все эти подэтапы подробнее будут описаны в дальнейшем.

Этапы 0° и 1° можно назвать постановочными, 2°—5° — реализующими, 6° — отладочным, 7°, 8°, 9° — заключительными. Первые 4 этапа проводятся без использования ЭВМ (или ее внешних устройств), последние (кроме этапа 9° и, может быть, 7°) — с ее использованием.

Все перечисленные этапы явно присутствуют и хорошо просматриваются при разработке достаточно сложных программ. Для простых задач некоторые этапы могут совмещаться друг с другом или проходить незаметно без какой-либо четкой фиксации. Например, для элементарных задач первые два или даже три этапа часто отсутствуют или соединяются в один подготовительный этап. Этим, в частности, можно объяснить то, что начинающие программисты, обучавшиеся программированию на элементарных задачах, при переходе к составлению сложных программ оказываются не готовыми к разработке технического задания, проекта, общих алгоритмов. Кроме того, первые этапы могут быть значительно сокращены, если программист получает задание на программирование, содержащее уже общий алгоритм разрабатываемой программы. С другой стороны, для сложных задач может быть несколько подэтапов алгоритмизации, отладки (автономная, комплексная) и добавляется период опытной эксплуатации.

0.2. Неизбежность отладки

Начинающий программист, как правило, переоценивает свои возможности и, проводя разработку программы, исходит из того, что в ее программе ошибок не будет. А говоря про только что составленную им программу, готов уверять, что она на 99% правильна, и ее остается только для большей уверенности один раз пропустить на машине с какими-нибудь исходными данными. Исходя из такого представления о своих способностях, этот программист и строит свою работу над программой, и каждый неверный результат, каждая найденная ошибка вызывают у него изумление и считаются, конечно, последними. Вследствие такого подхода получение на машине надежных результатов по составленной программе откладывается на длительный и весьма неопределенный срок. Только приоб-

рета достаточный опыт, программист понимает справедливость древнего высказывания: «Человеку свойственно ошибаться». Оказывается, что практически невозможно составить реальную программу без ошибок, и почти невозможно для достаточно сложной программы быстро найти и устранить все имеющиеся в ней ошибки. Трудности программирования и отладки подчеркивает следующий популярный афоризм: «В любой программе есть по крайней мере одна ошибка». Таким образом, можно сказать, что наличие ошибок в только что разработанной программе это вполне нормальное и закономерное явление. А совсем ненормальным, из ряда вон выходящим фактом является отсутствие ошибок в программе, которая не была еще подвергнута тщательной отладке. Конечно, речь здесь идет о реальных, достаточно сложных программах.

Учитывая этот печальный закон, разумно уже при разработке программы на этапах алгоритмизации и программирования (2° и 3°) готовиться к обнаружению ошибок на стадии отладки (6°), принимать профилактические меры по их предупреждению. Например, уже на этапе 1°, когда выбирается общий метод решения задачи, следует разработать и основную стратегию отладки, общий план ее проведения (см. 1' на рис. 1). Необходимо уже на этой стадии разработки программы решить, каким путем можно будет в дальнейшем убедиться, что выбранный метод решения задачи обеспечивает правильные результаты для указанных в ТЗ диапазонов изменения величин и условий функционирования программы.

При разработке алгоритма программы решаются тактические вопросы проведения отладки, намечаются способы контроля отдельных блоков и приемы предстоящей локализации ошибок в них (см. 2'). Для этого проектируются контрольные примеры, по алгоритмам (блок-схемам) намечаются места и моменты необходимой отладочной печати и выбираются выводимые на печать данные, которые должны обеспечить возможность быстрой локализации ошибок при отладке (на этапе 6.2°). Разрабатывая алгоритм, следует, таким образом, учитывать, можно ли будет достаточно просто проконтролировать программу, составленную по выбранному алгоритму, и в случае, когда предвидятся большие затруднения, нужно отдать предпочтение другому, более выгодному для этапа отладки, алгоритму. Нужно всегда помнить, что главным критерием ценности программы является ее правильность, и для гарантирования такого свойства программы следует жертвовать другими показателями, такими, например, как скорость работы или требуемый объем памяти. Давно ушли в прошлое те времена, когда программу оценивали только по количеству команд в ней.

Вообще, можно считать, что именно на этапах алгоритмизации и программирования закладывается фундамент быстрой и успешной

отладки на этапе 6°. Чем более тщательно проведены этапы 1°, 2° и 3°, и, в частности, чем более детально разработан план отладки (см. 1'), чем больше внимания было уделено проверке составленного алгоритма и программы, а также применению отладочных средств для облегчения предстоящей отладки (см. 2° и 3°), и, в конечном счете, чем больше потрачено времени и сил на этапах 1°—5°, тем меньше времени потребуется на проведение самой отладки и тем скорее будут получены на ЭВМ результаты, которым можно будет доверять. Наоборот, стремление к максимально быстрому составлению программы с целью скорейшего начала (и окончания) ее отладки, приводит к обратному результату: отладка затягивается надолго, и получение достоверных результатов откладывается на длительный срок. Такая задержка вызывается тем, что уже на поздних этапах отладки (а иногда — и счета) вскрываются все новые и новые ошибки, допущенные на этапах алгоритмизации и программирования, и приходится тратить много времени на их обнаружение и исправление. Обратившись к рис. 1, можно сказать, что кратность внутреннего цикла (этапы 5° и 6°) становится слишком большой, и при этом захватываются и этапы 2° и 3°, а иногда и кратность внешнего цикла (от этапа 1° до 8°), оказывается отличной от единицы. Кроме того, каждое выполнение внутреннего цикла требует обращения к ЭВМ за получением новых результатов, на ожидание которых тратится обычно от нескольких часов до нескольких дней. Напрашивается вывод, что для ускорения отладки нужно сократить кратности указанных циклов, выходя на машину с программой уже тщательно проверенной и подготовленной заранее к отладке.

Напомним содержание подэтапов 1'—5', имеющих целью облегчить предстоящую отладку и сократить время ее проведения.

1'. Разработка общего плана проведения отладки, общей методики проверки правильности составленной программы, а также системы необходимых для отладки контрольных примеров.

2'. Проверка разработанных алгоритмов, выбор отладочных средств и определение контролируемых ими мест, участков, величин.

3'. Проверка составленной программы, реализация намеченного ранее плана использования отладочных средств для получения на ЭВМ необходимых при локализации ошибок тестовых результатов; изготовление эталонных результатов для тестов.

4'. Ввод, печать и сверка текста программы, перенесенной на внешние носители.

5'. Получение с помощью транслятора вспомогательных таблиц (например, таблицы перекрестных ссылок) и проверка их.

(8'. Тщательный контроль первых результатов, получаемых по новой программе.)

У начинающих программистов изложенный плановый подход к проведению отладки (см. 1' и 2') вызывает вначале трудности, поскольку им приходится разрабатывать план отладки для несуществующей пока программы. Но нет другого пути освоить этот эффективный способ, кроме как развивать в себе навыки планирования своей работы и предвидения особенностей предстоящей отладки программы по ее проекту и общим алгоритмам.

Чем на более ранней стадии разработки программист начинает заниматься вопросами отладки программы, тем меньше неприятных неожиданностей ожидает его в будущем. Надежды на то, что устранение ошибок из программы и получение правильных результатов произойдет как-то само собой, без затраты особых усилий, никогда не оправдываются. Вообще, оптимизм и самоуверенность для программиста на стадии разработки противопоказаны; они могут являться полезными только на стадии отладки при затяжной борьбе с очень глубоко скрытыми ошибками.

Примерное распределение между этапами общего времени, необходимого для разработки достаточно сложных программ, выглядит следующим образом [1, 9]:

0°—1°. Получение задания, составление проекта программы и общего плана отладки	10%
2. Разработка алгоритма (15%) и детального плана отладки	20%
3°. Программирование (5%) и изготовление тестов	15%
4°—5°. Препарация и первая трансляция	5%
6°. Отладка	40%
7°. Оформление программы.	10%

Приведенные цифры отражают тот факт, что в процессе разработки программы работы по доказательству (демонстрации) правильности разрабатываемой программы равнозначны работам по ее изготовлению (проектированию, алгоритмизации и написанию), что можно выразить следующей формулой:

разработка программы = изготовление + доказательство.

Поэтому программой следовало бы называть только такую программу, которая выдает правильные результаты, а то, что еще не прошло стадию доказательства правильности, является не программой, а ее полуфабрикатом. Изготовление такого полуфабриката, конечно, является делом несравнимо более легким, чем разработка настоящей программы (особенно если изготовитель и не думает о последующей отладке).

Конечно, для простых задач распределение времени между этапами будет несколько другим, за счет увеличения доли программирования по отношению к алгоритмизации и отладке. Но, как правило,

время, затрачиваемое на работы, связанные с отладкой, составляет около половины всего времени, необходимого на разработку программы. Поэтому вопрос минимизации времени, необходимого на отладку, имеет особое значение. К его решению можно подойти с двух сторон:

а) путем ускорения поиска и исправления ошибок, имеющих в программе (см. гл. 1, 2, 3);

б) путем уменьшения количества ошибок, допускаемых при разработке алгоритма и составлении программы (см. гл. 4, 5).

Первые главы содержат ставшие уже классическими приемы отладки программ в рамках традиционной методики программирования; последующие главы затрагивают более современный подход к разработке программ.

ГЛАВА I

КОНТРОЛЬ ПРОГРАММЫ

Во введении подэтап контроля программы (6.1°) характеризовался как этап решения задачи, целью которого является установление наличия ошибок в составленной программе или убедительная демонстрация их отсутствия. Если будет установлено, что ошибки в программе имеются, то на следующем этапе (6.2°) будет производиться их поиск. Поэтому в задачу контроля входит также получение еще и такой информации о характере работы программы, которая могла бы помочь в дальнейшем при поиске ошибок.

Но, как было отмечено выше, работа по обнаружению ошибок производится и до этапа отладки, на этапах алгоритмизации и программирования, препарации и трансляции. Разница состоит лишь в том, что на этапе отладки для выявления ошибок в программе и их поиска используется как текст программы, так и результаты ее проверочного выполнения, и на предыдущих этапах — только сам текст программы. Поэтому, прежде чем рассматривать подробно этап контроля программы, обратимся к способам выявления ошибок на начальных этапах разработки программы.

1.1. Контроль текста

Сначала рассмотрим «ручные» методы контроля текста программ (алгоритмов), которые проводятся за столом без использования ЭВМ (пп. 1.1.1—1.1.3), затем — машинные, с применением ЭВМ (1.1.4—1.1.6).

Можно различать три способа контроля текстов алгоритмов и программ без применения ЭВМ: *просмотр, проверка и прокрутка.*

1.1.1. Просмотр. Текст составленной программы (или алгоритма — для этапа алгоритмизации) внимательно просматривается (читается) на предмет обнаружения описок и смысловых расхождений с текстом алгоритма, по которому производилось программирование (или более детальная алгоритмизация). Помимо сплошного просмотр-

ра применяется и выборочный просмотр некоторых фрагментов программы. Например, имеет смысл отдельно просмотреть организацию всех циклов, чтобы убедиться в правильности операторов, задающих кратности циклов. Полезно просмотреть еще раз условия в условных операторах, аргументы в обращениях к процедурам (подпрограммам) и т. д.

1.1.2. Проверка. При проверке программы (или алгоритма) программист по тексту программы мысленно старается восстановить тот вычислительный процесс, который определяет программа, после чего сверяет его с требуемым процессом, т. е. с заданным в ТЗ, определенным в проекте или в алгоритме, по которому было произведено программирование. Поскольку приемы проверки программ являются делом сугубо индивидуальным, то делать какие-либо обобщения и давать советы весьма трудно.

Самое главное о чем всегда следует помнить, так это о том, что ошибки в проверяемой программе обязательно есть, и чем больше их будет обнаружено за столом, тем легче и быстрее пройдет предстоящий этап отладки программы на машине. На время проверки нужно постараться «забыть» о том, что должен делать проверяемый участок программы, и «узнавать» об этом по ходу его проверки. Только после окончания проверки участка и выявления тем самым его действительных функций можно «вспомнить» о том, что он должен делать и сравнить реальные действия программы с требуемыми. Полезно, найдя какую-либо специфическую ошибку, отойти от последовательной проверки и сразу узнать, нет ли таких же ошибок в аналогичных, в особенности уже проверенных, местах.

К трудностям проверки программы, особенно логических ее участков, можно отнести то, что поскольку сверять приходится, собственно говоря, не тексты алгоритмов и программ, а вычислительные процессы, ими определяемые, то проверка часто имеет не визуальный характер, а мысленный. Поэтому нужно отметить, что чем ближе текст проверяемой программы к тексту исходного алгоритма (или при проверке алгоритма — тексты алгоритмов двух соседних этапов алгоритмизации — см. в 0.1 п. 2°), тем легче производить проверку программы и тем больше будет обнаружено ошибок. Таким образом, увеличение количества этапов алгоритмизации должно приводить к значительному облегчению и тем самым к увеличению эффективности, а в конечном счете, к ускорению проверки и отладки, несмотря на увеличение объема производимых работ при пошаговой алгоритмизации.

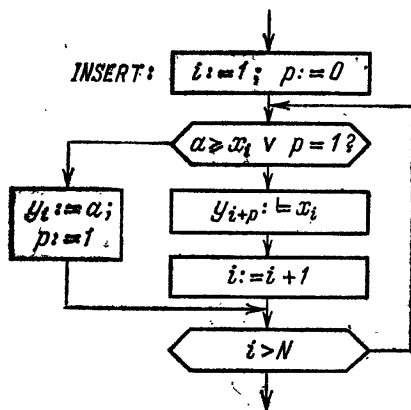
Все высказанные советы и соображения справедливы и для нижеследующего метода контроля программ.

1.1.3. Прокрутка. Другим способом контроля программ (алгоритмов) за столом является прокрутка (иногда ее называют «сухой»

прокруткой — *dry running* — для отличия от метода прокрутки, применяемого на этапе локализации, и использующего ЭВМ). Основой прокрутки является имитация программистом выполнения программы (алгоритма) на машине, с целью более конкретного и наглядного представления о процессе, определяемом текстом проверяемой программы. Прокрутка дает возможность приблизить последовательность проверки программы к последовательности ее выполнения, что позволяет проверять программу как бы в динамике ее работы, проверять элементы вычислительного процесса, задаваемого проверяемой программой, а не только статичный текст программы. Для выполнения прокрутки обычно приходится задаваться какими-то конкретными исходными данными и производить над ними необходимые вычисления.

Рассмотрим метод прокрутки на примере простой задачи.

Пример. Имеется массив x из N чисел, упорядоченный по неубыванию, и число a . Требуется образовать массив y из $N+1$ значений, объединив a и массив x так, чтобы не нарушить упорядоченность.



$a=3$	$a=1$	$a=5$
$x=\{2, 4\}$	$x=\{2, 4\}$	$x=\{2, 4\}$
$y=\{2, 3, 4\}$	$y=\{1, 2, 4\}$	$y=\{2, 4, ?\}$
$i=1 \setminus 2 \setminus 3$	$i=1 \setminus 2 \setminus 3$	$i=1 \setminus 2 \setminus 3$
$p=0 \setminus 1$	$p=0 \setminus 1$	$p=0$

Рис. 2. Пример прокрутки программы.

Предположим, что в результате разработки программы мы получили следующий алгоритм (см, рис, 2, сверху).

Для прокрутки нужно задаться какими-то конкретными значениями величин N , a и x_i . Видимо, не ограничивая общности, можно задать $N=2$; положим, кроме того, $a=3$ и массив $x=\{2, 4\}$. Выпишем на бумагу имена переменных, за которыми нам придется следить: a, x, y, i, p .

Выполняя первые два оператора, записываем полученные значения (1 и 0) рядом с переменными i и p (см. рис. 2, внизу). Далее, при проверке условия для текущего значения i получаем, что $a > x_1$ и поэтому затем будет выполняться $y_i := x_i$. Увеличиваем i на единицу (зачеркивая (\backslash) предыдущее его значение, равное единице), и, так как $i=N(=2)$, продолжаем выполнение программы. При выполнении цикла для $i=2$ получаем $a < x_2$ и $p \neq 1$; поэтому выполняем $y_2 := a$ и $p := 1$ (и зачеркиваем предыдущее значение p , равное нулю). Так как по-прежнему $i=N$, то продолжаем вычисления. Вследствие того, что $p=1$ на третьем цикле выполняем $y_3 := x_2$, увеличиваем i до 3 и, так как $i > 2 (=N)$, заканчиваем прокрутку алгоритма.

Анализируем полученные результаты: массив y получился правильным. На этом можно было бы и закончить ручной контроль программы, но стоит задуматься о том, будет ли алгоритм работать правильно, если задать другие значения исходных данных. На рис. 3 показаны результаты прокрутки для случаев, когда $a=1$ и $a=5$, т. е. рассмотрены случаи $a < x_1$ и $a > x_N$. Оказывается, что для последнего случая алгоритм срывает работу, так как элемент y_3 не получил никакого значения. Для исправления алгоритма можно в начало программы добавить оператор (блок) $y_{N+1} := a$, который необходим для случая $a > \text{Max } x_i (=x_N)$ и не мешает в остальных случаях, в чем можно убедиться, проведя повторные прокрутки.

Конечно, можно было бы найти ошибку в данном алгоритме и не прибегая к такой подробной прокрутке. Но для алгоритмов со сложной логикой, в которых, например, характер работы одного участка алгоритма зависит от результатов работы других его участков (см. действие величины p в примере), необходимо осуществлять прокрутку алгоритма для ряда подобранных специально исходных данных и параметров. Прокрутка дает программисту возможность найти более хитрые ошибки в алгоритме, чем при простой проверке.

Трудностью применения прокрутки является большой объем ручной работы при попытке точного моделирования работы программы. Поэтому успех применения прокрутки заключается в выборе необходимой степени детализации моделирования такой, чтобы, с одной стороны, выявить максимальное количество ошибок, а с другой стороны, затратить на это минимальные усилия. Приведем несколько соображений, которые могут помочь уменьшить время, затрачиваемое на прокрутку.

Прокрутку следует применять лишь для контроля логически сложных программ или блоков *). Арифметические блоки нужно проверять обычным способом, не задаваясь конкретными исходными данными. Вычислять числовые значения нужно лишь для тех величин, от которых зависит последовательность выполнения блоков (операторов) программы и эта последовательность является очень существенной. Поэтому во время прокрутки программы при каждой возможности, когда позволяет характер прокручиваемого блока программы, нужно переходить на обычную проверку, и возвращаться на режим прокрутки при начале проверки логически сложных блоков.

Исходные данные, влияющие на логику программы, должны выбираться такими, чтобы минимизировать прокрутку программы (см. $N=2$ в примере). Но с другой стороны данные должны быть таковы, чтобы в прокрутку вовлеклось большинство ветвей программы, чтобы прокрутка отразила типичный характер работы данной программы. Кроме того, в ходе прокрутки необходимо проверить работу программы и для особых случаев, для экстремальных значений параметра (например, для $N=1$ в предыдущем алгоритме). Многократные повторные прокрутки какого-либо участка программы можно не производить, если в логике его выполнения ничего не изменяется по сравнению с предыдущими прохождениями. Например, тело цикла можно прокрутить лишь для первых двух-трех проходов (проверка входа в цикл) и для последних одного-двух (проверка выхода из цикла).

Прокрутку можно существенно облегчить или сделать необязательной, если при алгоритмизации не стремиться чрезмерно к компактности программы и заранее позаботиться о простоте ее логического решения. Рассмотрим, например, следующую программу на языке PL/1, решающую ту же задачу, что и в приведенном только что примере.

```
INSERT1: DO I=1 BY 1 TO N WHILE (X(I) <= A);
                                         Y(I) = X(I); END;
      Y(I) = A; /* ЗНАЧЕНИЕ ПАРАМЕТРА
                  ЦИКЛА СОХРАНИЛОСЬ */
      DO I=1 BY 1 TO N; Y(I+1) = X(I); END;
```

*) Под блоком будем понимать как блоки алгоритмических языков (алгол-60, PL/1), так и просто некоторую группу операторов (блоков на блок-схемах), объединяемых по какому-либо признаку. Например, *арифметический блок* — выполняемая последовательно группа операторов, производящих вычисления в программе, *логический блок* — группа операторов, управляющих последовательностью вычислений в программе.

Программа состоит из трех выполняемых последовательно операторов, решающих свои частные задачи по формированию трех частей массива y : для $y_i < a$, $y_i = a$ и $y_i > a$. Очевидно, ввиду простоты логической структуры программы прокрутка ее будет значительно проще, так как сведется просто к последовательной проверке трех операторов; но следует убедиться в правильности работы программы для трех основных случаев соотношения исходных данных: $a < x_1$, $x_1 < a < x_N$; $a > x_N$. Упростить прокрутку нам помогло и то, что для записи алгоритма были использованы операторы PL/1 — более крупные и мощные операторы, чем те операторы блок-схемы, которые были применены в примере. Аналогичная программа на фортране имеет гораздо менее наглядный вид и, видимо, потребует настоящей прокрутки:

С ПРОГРАММА INSERTIF НА ФОРТРАНЕ

```

DO 3 I=1, N, 1
    K=I
    IF (.NOT. (X(I) .LE. A)) GO TO 5
3   Y(I)=X(I)
    K=K+1
5   Y(K)=A
    IF (K .GT. N) GO TO 7
    DO 6 I=K, N, 1
6       Y(I+I)=X(I)
7   . . . . .

```

Здесь в отличие от программы на PL/1 пришлось позаботиться о сохранении значения параметра цикла и об обходе последнего оператора цикла при $a > x_N$.

Кроме того, для сравнения приведем на PL/1 алгоритм, реализованный ранее в примере с помощью блок-схемы, но необходимым образом подправленный.

```

INSERT2: P=0; Y(N+1)=A;
DO I=1 BY 1 TO N;
    IF A >= X(I) ! P=1
    THEN Y(I+P)=X(I);
    ELSE DO; Y(I)=A; P=1; I=I-1; END;
END;

```

Пожалуй, для такой формы записи алгоритма прокрутка была бы еще более желательна, чем при использовании блок-схемы.

Прокрутка обычно является необходимой для программ, написанных в машинных командах или на языке ассемблера, а также для программ, имеющих дело со многими внешними носителями или использующих оперативную память сложным образом (например, с на-

ложением одних данных на другие). Осуществление прокрутки может потребовать больших листов бумаги для моделирования оперативной и внешней памяти и одновременного участия в работе 2—3 человек (один моделирует работу процессора, другой — оперативной памяти, третий — внешней памяти).

Прокрутка бывает необходимой и в том случае, когда программист не в состоянии вполне четко представить себе логику проверяемой программы, особенно если программа написана не им, и нет хорошего описания.

Прокрутка, как и проверка, применяется не только на этапах алгоритмизации (3°) и программирования (4°) для контроля только что составленного алгоритма или программы, но и в ходе отладки программы на ЭВМ (этап 6°) для повторного ее контроля за столом и локализации ошибок. Различие в прокрутке на этапах разработки программы и ее отладки заключается в том, что на этапе отладки прокручивается не вся программа, а лишь подозреваемые места, но прокрутка производится более детально и тщательно, так как направлена против глубоко скрытых ошибок, которые не обнаруживаются другими способами.

Несмотря на все трудности применения прокрутки, ориентация на нее для поиска ошибок, содержащихся в составленной программе, может оказаться более эффективным способом, быстрее приводящим к цели, чем попытка обнаружить большинство ошибок только по результатам тестовых прогонов программы. Дело в том, что в случае достаточно большого количества ошибок в программе или в отдельных ее блоках, их влияние на полученные тестовые результаты становится трудно предсказуемым, и их обнаружение, тем самым, весьма затруднительным. Поэтому желательно до начала поиска ошибок по тестовым результатам обнаружить максимальное количество ошибок по тексту, прибегнув к такому трудоемкому, но эффективному способу, как прокрутка.

Можно рекомендовать применять прокрутку всего алгоритма программы после этапа алгоритмизации, а после этапа программирования сосредоточить все внимание на проверке правильности перевода алгоритма в программу; прокрутка программы производится лишь для отдельных фрагментов и уже в ходе локализации ошибок.

В заключение необходимо отметить, что использование прокрутки оказывает положительное влияние на предстоящее проведение отладки и в силу того, что содействует глубокому осознанию программистом действительной логики составленной им программы и того реального вычислительного процесса, который ею задается. Ведь быстрота ориентирования в отлаживаемой программе и в выдаваемых отладочных результатах всецело зависит от способности

программиста мысленно представить себе во всех деталях алгоритм рассматриваемой программы, что невозможно без глубокого и крепкого знания его в течение всего длительного времени проведения отладки.

1.1.4. Печать текста. Для проверки правильности препарации программы (переноса текста программы на какие-либо машинные носители) производится ввод в машину и выдача на печать текста программы для последующей его сверки с исходным текстом на бланках.

Несмотря на простоту такого способа контроля программы, он позволяет выявить за один выход на машину обычно сразу множество ошибок. В противном случае, может быть, пришлось бы затратить много времени на обнаружение ошибок, так как на этапе отладки на каждую ошибку в среднем тратится не менее одного выхода на машину.

1.1.5. Трансляция. Трансляция позволяет выявить синтаксические ошибки, имеющиеся в программе. Кроме того, от транслятора можно получить и другую информацию о программе, полезную во время ее проверки. Например, в ОС ЕС можно получить таблицу используемых идентификаторов, задав опцию ATR (для PL/1) или MAP (для фортрана); в таблице ATR дополнительно печатаются атрибуты (описатели) используемых в модуле величин и номера операторов, где они описаны. Кроме того, для транслятора PL/1 можно выдать таблицу перекрестных ссылок, задав опцию XREF транслятора. Таблица содержит идентификаторы всех величин, используемых в транслируемом модуле, с указанием номеров операторов, в которых встречаются эти величины. Особенно полезна такая таблица для программ, написанных на PL/1 или фортране, в которых вследствие правил умолчания допускаются не описанные явно идентификаторы, и, кроме того, нет средств для обнаружения случая использования в выражении переменной, не получившей еще никакого значения. Поэтому из-за описок или ошибок препарации в программе свободно могут появляться (подобно тому, как появился поручик Кижэ) новые фиктивные переменные, замещающие в некоторых местах транслируемого модуля реальные величины и не обнаруживаемые системой ни при трансляции, ни при счете. Например, если в последнем приведенном примере INSERT 2 в первом его операторе вместо P написано R, то программа будет выполняться, и в некоторых случаях даже получатся правильные результаты (если в ячейке памяти отведенной для P окажется нулевое значение).

Для выявления таких таинственных переменных и применяются опции транслятора ATR, XREF (в PL/1) и MAP (в фортране); в алголе ЕС таблица идентификаторов выдается вместе с распечаткой транслируемой программы.

Например, директива вызова транслятора может иметь такой вид:

```
//SHAG EXEC PLILFC,PARM.PLIL='ATR,XREF' ДЛЯ PL/I  
//STEP EXEC FORTGC,PARM.FORT='MAP' ДЛЯ ФОРТРАНА
```

1.1.6. Статический анализ. В некоторых трансляторах или пакетах прикладных программ имеются средства, которые выдают информацию о строении транслируемой программы в виде блок-схемы, которую можно сопоставить с блок-схемой, принятой в качестве основы при программировании. По полученной блок-схеме можно выявить некоторые ошибки в строении проверяемой программы, в частности, обнаружить никогда не выполняемые участки программы, отсутствие разветвления в нужном месте или неправильную связь блоков и т. п.

1.2. Контроль результатов

Как бы ни была тщательно проверена и прокручена программа за столом, решающим этапом, устанавливающим ее пригодность для работы, является контроль программы по результатам ее выполнения на ЭВМ. Для каждого класса задач могут существовать свои особые, предназначенные только для этих задач, способы контроля программ на ЭВМ. Мы рассмотрим здесь лишь один метод контроля, который можно считать универсальным для всех классов задач — метод контрольных тестов (test — испытание). Тестом будем называть информацию, состоящую из исходных данных, специально подобранных для отлаживаемой программы, и из соответствующих им эталонных результатов (не только окончательных, но и промежуточных), используемых в дальнейшем для контроля правильности работы программы.

1.2.1. Тестирование. При использовании метода тестов программа (или ее отдельный блок) считается правильной, не содержащей ошибок, если пропуск отлаживаемой программы для выбранной системы тестовых исходных данных во всех случаях дает правильные результаты. Таким образом, контроль программы сводится к тому, чтобы подобрать такую систему тестов, получение правильных результатов для которой гарантировало бы правильную работу программы и для остальных исходных данных из области, указанной в техническом задании. Для реализации метода контрольных тестов должны быть изготовлены или заранее известны эталонные результаты, на основании сверки с которыми получаемых тестовых результатов, можно было бы сделать вывод о правильности работы программы на данном тесте. Эталонные тестовые результаты для вычислительных задач можно получить, осуществляя вычисления вручную, при-

меняя результаты, полученные ранее на другой ЭВМ или по другой программе, или используя известные факты, свойства, физические законы. Можно считать, что имеется 3 способа тестирования: алгоритмическое, аналитическое, содержательное.

Алгоритмическое тестирование, которое нас интересует в первую очередь, применяется программистом для контроля этапов алгоритмизации и программирования. Программисты проектируют тесты и начинают готовить эталонные результаты на этапе алгоритмизации, а используют их на этапе отладки (6.1° и 6.2°).

Функциональное или аналитическое тестирование служит для контроля выбранного метода решения задачи, правильности его работы в выбранных режимах и с установленными диапазонами данных. Тесты проектируют и начинают готовить на этапе проектирования сразу после выбора метода, а используют их на последнем этапе отладки или для анализа результатов пробного счета (этап 8'); в ходе тестирования, наряду со сверкой на совпадение, применяются и качественные оценки результатов.

Содержательное тестирование служит для проверки правильности постановки задачи. Для контроля при этом используются, как правило, качественные оценки и статистические характеристики программы, физический смысл полученных результатов и т. п. В проведении содержательного тестирования, принципы которого формулируются в техническом задании, самое активное участие должны принимать заказчики или будущие пользователи программы.

Содержательные и аналитические тесты проверяют правильность работы программы в целом или крупных ее частей, в то время как алгоритмические тесты в первую очередь должны проверять работу отдельных блоков или операторов программы. Алгоритмические тесты помимо установления наличия ошибок в программе должны, по возможности, давать определенную информацию также и о местонахождении ошибок в программе, т. е. должны помогать локализации ошибок на следующем этапе (6.2°). В дальнейшем мы будем иметь в виду, как правило, только алгоритмические тесты.

Разрабатывая систему тестов, нужно стремиться к тому, чтобы успешный пропуск ее на машине доказывал отсутствие ошибок в программе (или отдельном ее блоке), хотя для многих достаточно сложных программ, особенно, если над ними работает несколько программистов, можно практически говорить лишь о большей или меньшей уверенности в правильности программы. Это объясняется тем, что изготовление и пропуск *всех* тестов, необходимых для доказательства, может потребовать такого объема работ, который затянет этап контроля на многие месяцы или годы. Поэтому при разработке системы тестов наряду с задачей всестороннего и глубокого тестирования, стоит задача минимизации количества необходимых

тестовых результатов, машинного времени и усилий программиста. В большинстве случаев при использовании метода тестирования вопрос о доказательстве отсутствия ошибок практически можно ставить лишь для небольших блоков (модулей) программы, а для целой программы приходится ограничиваться той или иной вероятностью отсутствия ошибок в программе. Примером очень трудно находимой ошибки в программе является случай, когда какой-либо модуль (блок) помимо того, что он правильно реализует все необходимые функции, делает и что-то лишнее, что нарушает работу всей программы после стыковки модулей (например, непредвиденно изменяет значение аргумента, передаваемого в подпрограмму именем, портит общую или базированную величину).

1.2.2. Правила тестирования. Здесь рассмотрены некоторые правила тестирования, в которых делается попытка учесть как желательность доказательства правильности контролируемой программы, так и ограниченность человеческих возможностей при проведении такого доказательства.

Проход участков. Каждый линейный участок программы должен быть обязательно пройден при выполнении, по крайней мере, одного теста. Очевидно, что в противном случае никакой гарантии о правильности работы всей программы нельзя будет дать.

Часто бывает трудно с помощью задания исходных данных для программы управлять прохождением в ней определенных участков. Поэтому практически речь может идти лишь о выполнении этого правила для поблочного контроля, т. е. о задании таких тестовых данных для отдельных блоков программы, которые бы направляли процесс выполнения блока по нужным путям и участкам. Здесь встает вопрос об автоматизации учета пройденных (и непройденных) участков программы, но никаких конкретных специальных средств пока предложить нельзя; средства слежения и прокрутки (см. 2.1) в данном случае мало пригодны.

Как правило, с помощью специально подобранных тестовых входных данных (может быть, и не вполне реальных в данной задаче) можно проверить работу отдельных блоков гораздо более надежно, чем с реальными данными, которые блоки получают от соседних блоков во время их совместной работы. Кроме того, такие условные данные позволяют обходить выполнение (и контроль) протестированных уже ранее участков или блоков, и тем самым избегать дублирования работы и бесполезной траты времени. Но такой независимый контроль работы отдельных блоков бывает возможен лишь в том случае, если работа самих блоков мало зависит друг от друга, т. е., если проверяемые блоки обладают свойством модульности, которое закладывается еще при разработке алгоритма (см. ниже гл. 4). В том случае, когда выполнение некоторого участка меняет по-

рядок выполнения или характер работы других участков (см. выше пример прокрутки), может потребоваться многократная проверка участков программы, и даже перебор всех ветвей программы (т. е. проход по всем возможным путям выполнения программы); многократная проверка требуется, в частности, и для участков, содержащих переменные с индексами.

Если вернуться к рассмотрению примеров, приведенных выше, то для того, чтобы при выполнении программ INSERT1 и INSERT2 было выполнено сформулированное требование и пройдены оба участка, достаточно задать a , удовлетворяющее условию $x_1 \leq a < x_N$. Но в обоих случаях пропуск только одного такого теста не гарантирует правильности контролируемой программы: из-за специфики реализующих программ их необходимо также проверить на случай $a > x_N$. Для программы INSERT1 это необходимо сделать из-за того, что оператор цикла проверен не полностью: окончание цикла по N для первого теста осталось непроверенным (выход из цикла произошел по WHILE). Для программы INSERT2 второй прогон необходим для проверки оператора предварительной засылки a в y_{N+1} . Ввиду логической сложности алгоритма реализованного в INSERT2 не лишним будет проверить и работу программы на других тестах, в частности для $a < x_1$.

Точность проверки. Контроль арифметических блоков (как и других блоков) производится путем сверки результатов, полученных при выполнении блока, с эталонными результатами. Для арифметических результатов дополнительная сложность заключается в определении точности, с которой необходимо сверять (и, тем самым, вычислять) эталонные и тестовые результаты, с тем чтобы можно было действительно удостовериться в правильности работы блока. Трудность заключается в том, что величины, входящие в некоторое проверяемое арифметическое выражение, встречающееся в блоке, в зависимости от соотношения их значений и характера производимых над ними операций вносят различный вклад в результат. Поэтому может оказаться, что неправильно запрограммированное выражение для некоторых тестовых значений величин, входящих в него, будет иметь якобы правильное значение ввиду того, что результат неправильной операции или неверно вычисленный ранее операнд выражения не окажут почти никакого влияния на тестовое (сравниваемое) значение выражения.

Например, для оператора $c := a + b$ из того, что c совпало с эталонным значением, не следует, что выражение реализовано в программе верно, поскольку для случая, когда $a \gg b$, замена знака плюс на минус не была бы обнаружена, если бы эталонное значение c было вычислено с недостаточной точностью. Кроме того, если вычисление a и b не было проверено ранее, то из правильности c нельзя

сделать вывод о правильности реализации вычисления b (для случая $a \gg b$).

В операторе

$$d := a \times (c - x) + b$$

при близких значениях c и x , для того чтобы по значению d убедиться в правильности реализации выражения правой части, а также входящих в него величин, придется вычислять эталонный результат для d и сравнивать его с тестовым результатом с очень большой точностью (случай $a \times (c - x) \ll b$).

В выражении $a/(c - x) + b$ при близких значениях c и x трудно проверить правильность значения переменной b и операции, выполняемой над ней. Но если подобрать такие исходные данные, чтобы получилось $c \gg x$ (или $c \ll x$), то будут трудности с проверкой правильности x (или c) по вычисленному значению всего выражения.

Таким образом, для того чтобы быть уверенным в том, что правильный числовой результат, полученный на машине, говорит о правильности программы, необходимо следить за промежуточными результатами вычислений, которые не должны выходить за определенный диапазон, устанавливаемый в зависимости от точности вычислений эталонных результатов. Выполнение такого требования может привести к необходимости многократной проверки выражения для различных диапазонов данных.

Минимальность вычислений. Когда продолжительность работы контролируемой программы и, тем самым, количество вычислений и необходимых для контроля тестовых данных зависит от каких-либо параметров, то при контроле (как и при сухой прокрутке — см. выше) их следует выбирать такими, чтобы они минимизировали количество вычислений. К таким параметрам, например, могут относиться шаг или отрезок интегрирования, порядок матрицы или количество элементов вектора, длина символьных строк, точность для итерационных вычислений и т. п. Конечно, такая минимизация не должна значительно снижать надежность контроля. Следует заметить также, что значения исходных данных нужно выбирать такими, чтобы изготовление эталонных результатов вручную было, по возможности, облегчено. Например, данные могут быть сначала взяты целочисленными или такими, чтобы при проверке выражений некоторые их слагаемые, уже проверенные ранее, обращались в нуль.

Достоверность эталонов. Нужно обратить внимание и на достоверность процесса получения эталонных результатов. По возможности они должны вычисляться не самим программистом, а кем-то другим, с тем чтобы одни и те же ошибки в понимании задания на программирование или алгоритма не проникли и в программу и в эталонные результаты. Если тесты готовит сам программист, то эта-

лоны нужно вычислять до получения на машине соответствующих результатов, по крайней мере, до начала их использования при сверке. В противном случае имеется опасность невольной подгонки вычисляемых значений под желаемые, полученные ранее на машине. В качестве эталонных результатов можно использовать и данные, полученные при прокрутке программы.

Планирование. Основой эффективного тестирования является его плановость и систематичность. Только действуя по плану, заранее составленному на этапах проектирования и алгоритмизации (см. 1' и 2' во введении), учитывающему структуру и особенности разрабатываемой программы, можно надеяться быстро выявить все (или почти все) ошибки в программе и продемонстрировать с большой долей убедительности правильность тестируемой программы.

При плановом подходе проверяется блок за блоком (для алгоритмического тестирования), режим работы программы — за режимом (для функционального и содержательного контроля). Намечается как общий подход к контролю программы и основные его принципы (стратегия контроля), так и особенности реализации выбранных принципов, например, решается задача минимизации количества тестов и выходов на машину (тактика контроля). Таким образом, заранее устанавливается что необходимо проконтролировать и как это лучше выполнить. Планы контроля намечаются как для всей программы, так и для отдельных ее модулей, блоков и даже операторов, т. е. для тестирования как в крупном масштабе, так и в мелком.

Если программа состоит из центрального блока, который проводит обращения к периферийным блокам, мало связанным друг с другом, то возможны следующие два основных подхода к контролю такой программы, два основных направления тестирования: от периферии к центру (восходящий контроль) или, наоборот, от центра к периферии (нисходящий контроль). При первом, *восходящем*, способе, применяемом обычно для небольших программ, сначала контролируют отдельные периферийные блоки, а затем переходят к контролю центральной части, которая, таким образом, будет взаимодействовать только с отлаженными уже блоками.

При втором, *нисходящем*, способе, используемом для достаточно больших программ, параллельно с контролем периферийных блоков (или даже до начала их контроля) производится и контроль центрального блока, выполняемого на машине совместно с имитаторами периферийных блоков, называемых иногда «затычками» («заглушками»). В задачу имитаторов входит моделирование работы соответствующих блоков, с целью поддержать функционирование центрального блока. Обычно имитаторы выдают простейший результат, например, константу и сообщение о факте своего участия

в работе. Вместо постоянной величины на более поздней стадии отладки может выдаваться и случайная величина в требуемых диапазонах. Например, для начального контроля программы, включающей в качестве одного из своих блоков вычисление определенного интеграла, имитатор такого блока может выдавать константу или величину, получаемую по некоторому простейшему или ожидаемому закону. В свою очередь, блок интегрирования сам имеет периферийный блок вычисления подинтегральной функции, в качестве имитатора которой, поначалу также может быть взята константа или простейшая функциональная зависимость.

Практически, оба эти способа редко используются в чистом виде, отдельно один от другого. Обычно ко времени, когда приступают к контролю центрального блока, какие-то простейшие периферийные блоки уже отлажены автономно, и нет необходимости моделировать их работу и разрабатывать имитаторы. Кроме того, имитация некоторых других блоков оказывается делом очень трудоемким и поэтому начало контроля центральной части приходится согласовывать с окончанием отладки таких блоков.

Преимуществом ранней отладки центрального блока при нисходящем тестировании является то, что программист быстро получает возможность проверить периферийные блоки в условиях, которые в необходимой степени приближены к реальным. Действительно, центральный блок, снабженный хотя бы и простейшими функциональными возможностями, можно рассматривать как реальную среду, в которую погружаются вновь отлаживаемые блоки, добавляемые к центральной части. Добавление отлаживаемых блоков удобно производить по одному для быстрой локализации ошибок, возникающих при стыковке с центральным блоком. Подключение каждого нового блока к центральной части позволяет постепенно усложнять испытания, которым подвергается тестируемая программа.

Постепенность усложнения тестирования необходимо соблюдать и при восходящем способе, что позволит быстрее обнаруживать ошибки, которые будут проявляться в выдаваемых результатах не все сразу, взаимодействуя одна с другой, а по очереди.

При отсутствии планового подхода тестирование, обычно, сводится к тому, что программист берет какие-то, можно сказать, первые попавшиеся исходные данные и пропускает программу многократно, исправляя ее при обнаружении ошибок и добиваясь того, чтобы получаемые результаты походили на желаемые. При этом контролируется только некоторая часть блоков и операторов, а остальные выполняются в 1-й раз уже во время счета, и будут ли при этом найдены ошибки, имеющиеся в них, зависит только от случая.

Впрочем, для очень сложных программ (или блоков) планом может быть установлена и такая последовательность контроля, при которой сначала проверяется работа всей или некоторой части программы для типовых исходных данных, а затем, когда появляется уверенность, что программа в основном работает правильно, начинается ее детальное тестирование с использованием и условных данных. Такой подход весьма экономичен и удобен для начинающих программистов, так как позволяет им постепенно втягиваться в детали тестирования. Опасность заключается в том, что после успешного окончания основного тестирования «на радостях» обычно забывают о необходимости дальнейшего и более тщательного контроля программы и отдельных ее участков, да и настроиться на такой контроль становится уже психологически трудно.

Генерация тестовых данных. Если программа или какие-либо ее блоки активно используют в своей работе внешнюю память, то при их отладке удобно применять генератор тестовых данных, с помощью которого программист может записать на магнитную ленту или диск данные, предназначенные для отладки блоков и используемые этими блоками в качестве исходных. Записываемые генератором данные получаются из заданных программистом данных путем их размножения и преобразования по правилам, указанным в управляющей информации.

Примером генератора тестовых данных является служебная программа IEBDG в ОС ЕС ЭВМ.

1.2.3. Типы тестов. Тот вид контроля, который рассматривался до сих пор, можно назвать тестированием основных функциональных возможностей программы — *основной тест*. Помимо основного теста можно выделить следующие виды тестов.

Вырожденный тест. Этот тест затрагивает работу отлаживаемой программы в самой минимальной степени. Обычно тест служит для проверки правильности выполнения самых внешних функций программы, например, обращения к ней и выхода из нее.

Тест граничных значений. Тест проверяет работу программы для граничных значений параметров, определяющих вычислительный процесс. Часто для граничных значений параметра работа программы носит особый характер, который, тем самым, требует и особого контроля. Например, полезно проконтролировать работу программы п. 1.1.3 при $N=1$.

Аварийный тест. Тест проверяет реакцию программы на возникновение разного рода аварийных ситуаций в программе, в частности, вызванных неправильными исходными данными. То есть проверяется диагностика, выдаваемая программой, а также окончание ее работы или, может быть, попытка исправления неверных исходных данных.

Помимо автономных тестов, предназначенных для контроля отдельных блоков программы, можно выделить стыковочные и комплексные тесты. *Стыковочные тесты* предназначены для проверки взаимосвязи (стыковки) уже отлаженных частей программы. *Комплексные тесты* проверяют правильность работы всех или большинства частей программы после их объединения.

1.3. Классификация методов контроля

Ниже приведена схема, отражающая ту классификацию методов контроля, которая была принята выше.

Контроль:

1. По тексту.

1.1. Без ЭВМ.

1.1.1. Просмотр.

1.1.2. Проверка.

1.1.3. Прокрутка.

1.2. С ЭВМ.

1.2.1. Печать.

1.2.2. Трансляция (синтаксический контроль).

1.2.3. Статический анализ.

2. По результатам.

2.1. Тестирование.

2.1.1. Алгоритмическое.

2.1.2. Функциональное.

2.1.3. Содержательное.

2.2. Специальные методы.

Дополнение к 2.1 (по другому признаку):

2.1.1. Автономное.

2.1.2. Стыковочное.

2.1.3. Комплексное.

ГЛАВА 2

ЛОКАЛИЗАЦИЯ ОШИБОК

После того как с помощью контрольных тестов (или каким-либо другим путем) установлено, что в программе или в конкретном ее блоке имеется ошибка, возникает задача ее локализации, то есть установления точного места в программе, где находится ошибка.

Можно считать, что процесс локализации ошибок состоит из следующих трех основных компонент: (1) получение на машине тестовых результатов; (2) анализ тестовых результатов и сверка их с эталонными; (3) выявление ошибки или формулировка предположения о характере и месте ошибки в программе. Если ошибка найдена, производится ее исправление (см. гл. 3); в противном случае осуществляется переход к следующему шагу процесса локализации, то есть к получению дополнительных тестовых результатов.

В ходе поиска ошибок программист, анализируя полученные на машине результаты, проверяет различные предположения о характере и месте ошибки в программе, которые при этом приходят ему в голову. В случае несоответствия этих гипотез выданным результатам, программист выдвигает новые гипотезы и проверяет их или в уме, или проводя вычисления за столом, или обращаясь за новыми результатами к машине.

В таком характере работы программиста можно найти нечто общее с расчетом вариантов, который осуществляет шахматист (или шашкист) во время игры, когда он путем расчетов в уме ищет выигрывающий ход в позиции на шахматной доске, подвергая проверке один из заслуживающих внимания ходов за другим. Не найдя выигрывающего хода, шахматист делает какой-то, по его мнению, хороший ход, приближающий его к цели. Так и программист, не найдя ошибки путем исследования полученных тестовых результатов, делает новое предположение о месте или о характере ошибки, вставляет новую отладочную печать или изменяет программу («ход программиста»), а машина выдает новые тестовые результаты («ход машины»). Машина выступает как своеобразный партнер, задача которого заключается в том, чтобы вскрыть ошибки в рассуждениях программиста, как бы

сформулированных им в тексте отлаживаемой программы. Продолжая аналогию, можно сказать, что подобно тому, как нельзя реально надеяться выиграть партию в два—три хода, так же нельзя найти все ошибки в реальной программе за одно—два обращения к машине.

Программистов, успешно проводящих поиск ошибок в программе, можно условно разделить на «аналитиков» и «экспериментаторов». Аналитики отлаживают программу, редко выходя на машину и обходясь простейшими способами получения тестовых результатов на машине, путем тщательного изучения этих результатов и на основании глубокого и четкого представления о структуре и особенностях алгоритма отлаживаемой программы. Экспериментаторы ищут ошибки, изошренно используя всевозможные отладочные средства, быстро получая необходимые для все большей и большей локализации ошибок промежуточные результаты и легко ориентируясь в них. Конечно, идеальным является случай, когда программист сочетает в себе способность к глубокому расчету в уме различных вариантов работы программы и навыки работы с разнообразными отладочными средствами (для быстрого получения необходимых сведений о реальном вычислительном процессе, реализуемом отлаживаемой программой).

Если успех аналитического подхода к поиску ошибок зависит, видимо, от способностей и опыта программиста, то изучение и использование средств, помогающих локализации ошибок — главным образом средств получения необходимых промежуточных результатов — доступно каждому программисту. Имеются в виду средства, которые существуют в используемом языке программирования, в операционной системе, в специальных пакетах прикладных программ. Причем под промежуточными результатами выполняемой программы договоримся понимать как *арифметические* результаты, характеризующие значения используемых величин (в том числе, например, и строчных), так и *логические* «результаты», т. е. информацию, содержащую сведения о факте или последовательности выполнения операторов программы.

Далее рассматриваются следующие, ставшие уже классическими, способы получения программистом промежуточных результатов, вырабатываемых отлаживаемой программой:

- 1) аварийная печать (dump — остатки, отходы);
- 2) печать в узлах (snapshot — моментальный снимок);
- 3) слежение (tracing, trace — след);
- 4) прокрутка (running — просматривание).

В разделе 2.1 излагается основное назначение перечисленных способов, а также те средства языков PL/1, фортрана, алгола и соответствующих трансляторов в ОС ЕС ЭВМ, которые позволяют обеспечить использование названных способов в ходе отладки. Изложе-

ние средств локализации производится на примерах, а для детального ознакомления с используемыми возможностями следует обратиться к имеющейся литературе [10—15]. Методика применения отладочных средств, а также некоторые приемы и принципы локализации ошибок излагаются в 2.2 и 2.3.

2.1. Средства локализации

В PL/1 основой при реализации рассматриваемых средств локализации являются исключительные ситуации и связанный с ними оператор ON. В фортране ЕС такой основой является отладочный пакет, а в алголе ЕС — отладочные опции транслятора. Кроме средств языков и трансляторов имеются и общесистемные средства отладки.

2.1.1. Аварийная печать.

Под *аварийной печатью* понимается печать значений переменных в программе в тот момент ее выполнения, когда в ней возникает ошибка, препятствующая дальнейшему нормальному ее выполнению — *авария*; обычно после осуществления такой печати выполнение программы прекращается.

Благодаря аварийной выдаче программист получает доступ к тем значениям переменных, которые они имели в момент возникновения аварийной ситуации. Изучение и сопоставление таких значений обычно позволяет программисту достаточно точно локализовать ошибку в программе, а иногда и не одну.

Но во время осуществления аварийной печати могут выдаваться значения и тех переменных, которым еще ничего не было присвоено; в этом случае на печать выводятся случайные значения из ячеек памяти, отведенных под такие переменные. Для того, чтобы предотвратить возникновение ошибок печати при выводе таких значений и для облегчения разбора аварийной выдачи, всем переменным, используемым в программе, следует при их описании или в самом начале блока присваивать какие-либо специфические значения. В PL/1 и фортране предусмотрены специальные средства для облегчения такого присваивания (атрибут INITIAL для PL/1 и указания начальных значений в «разрезах» / и / для фортрана). Например:

```
DECLARE R(0:M) FIXED INITIAL ((M+1)55555);
```

или

```
REAL * 8 A(100)/100 * 8888/, B (5, 7) /35 * 88888888/
```

В фортране для инициализации общих переменных используется подпрограмма-спецификация BLOCK DATA.

PL/1. В программе на PL/1 аварийную печать можно осуществить оператором

ON ERROR PUT DATA (список идентификаторов); где в скобках через запятую перечисляются идентификаторы тех переменных, значения которых требуется выдать на печать в случае возникновения аварийной ошибки.

Напомним, что оператор ON действует только на те операторы охватываемого ON-оператора блока (и вложенных блоков), которые выполняются после выполнения данного оператора ON, но до выполнения следующего оператора ON. Это дает возможность для ошибок, возникающих в разных блоках или разных местах одного и того же блока, задавать печать различных переменных.

Рассмотрим, например, программу со следующим строением (ввиду условности примера атрибут INITIAL опущен):

```
P: PROCEDURE OPTIONS(MAIN);  
  DECLARE A(1:300);  
  . . . . .  
  S1: ON ERROR PUT DATA (A);  
  . . . . .  
  Q: BEGIN; DECLARE B, C (0:40, 0:10), N;  
    . . . . .  
    S2: ON ERROR PUT DATA (B);  
    . . . . .  
    T: PROCEDURE;  
      DECLARE A, B (0:N, 0:N);  
      . . . . .  
      S3: ON ERROR PUT DATA (N);  
      . . . . .  
      END T;  
      S4: ON ERROR PUT DATA (C);  
      CALL T;  
      . . . . .  
    END Q;  
  . . . . .  
END P;
```

Если предположить, что ON-операторы выполняются в указанном порядке, то в случае, если ошибка произойдет до выполнения оператора ON с меткой S1, никакой аварийной печати не происходит, и выполнение программы заканчивается с выдачей диагностического сообщения от системы. Для ошибки, возникшей в операторах, выполняемых между ON-операторами и метками S1 и S2, производится печать значения величины A. При ошибке в операторах от метки S2 до S4, принадлежащих блоку Q и не входящих в тело процедуры T, происходит печать B, а после оператора с меткой S4 — печать C. В процедуре T ошибка в операторах, следующих за оператором ON, приводит к печати N. Ошибка в операторах, стоящих вслед за оператором END Q, снова приводит к печати A.

Помимо печати, управляемой данными (PUT DATA), могут быть использованы, конечно, и другие виды печати, но приведенный способ обычно удобнее при отладке, так как облегчает разбор печати, хотя для экономии бумаги можно иногда воспользоваться и PUT EDIT. Аварийная печать может определяться и группой операторов, задаваемых внутри блока, входящего в ON-оператор.

В качестве оператора печати можно задать и оператор вида PUT DATA; без списка имен переменных. В этом случае будет производиться печать значений всех переменных, известных в том месте программы, где находится оператор ON (а не в том месте где произошла ошибка). Будут выдаваться также и значения переменных, описанных во внешних блоках; выдаются даже переменные, имена которых совпадают с именами переменных в блоке, где находится оператор ON. Например, если в приведенной выше программе все ON-операторы имеют вид

ON ERROR PUT DATA;

то при ошибке в программе по оператору с меткой S1 будет печататься значение A; по оператору S2 — B, C, A, N; по оператору S3 — A, B из блока T и B, C, N из блока Q, а также A из блока P; по оператору S4 — B, C, N и A.

Если в программе оставить только один ON-оператор с меткой S1, то при возникновении ошибки в любом случае будет производиться печать A.

Таким образом, пример показывает, что чем в более внутреннем блоке стоит оператор ON, тем больше переменных будет выдаваться на печать; разумеется, печататься будут только результаты работы блоков, активных в момент возникновения ошибки. При разборе такой печати может быть полезна таблица идентификаторов, выдаваемая транслятором (параметр трансляции — ATR).

Для того, чтобы аварийную печать можно было легко отключить, ее следует задавать по условию (см. ниже общее замечание).

З а м е ч а н и я. 1. Ситуации SIZE, SUBSCRIPTRANGE, STRINGRANGE по умолчанию выключены, и поэтому для выполнения аварийной печати при возникновении этих ситуаций их необходимо включить с помощью соответствующих префиксов. В противном случае трудно будет найти ошибки, связанные с выходом значения переменной за установленный диапазон, значения индекса — за его объявленные границы, подстроки — за строку, из которой она выделяется.

2. Для отладки группы операторов, реализующих саму аварийную печать, можно вызвать ситуацию ERROR, задав в нужный момент оператор

SIGNAL ERROR;

3. Одним из вариантов аварийной печати можно считать «финальную» (или «финишную») печать, производимую перед самым окончанием работы программы (аварийным или нормальным). В PL/1 такая печать осуществляется с помощью ситуации FINISH по оператору вида:

ON FINISH печать;

Финальная печать отличается от аварийной тем, что она (при правильном размещении ON-операторов PL/1) гарантирует выдачу значений выбранных (или всех) переменных, используемых в программе: печать производится или при возникновении аварии в программе, или, в случае отсутствия аварии, после выполнения последнего оператора программы.

4. Если при аварийной выдаче по оператору PUT DATA; выдаются на печать переменные, которым еще не было присвоено значение, может вновь возникнуть ошибочная ситуация. Для предупреждения возможного заикливания в этом случае ON-оператор должен иметь следующий вид:

ON ERROR BEGIN; ON ERROR SYSTEM; PUT DATA; END;

Фортран. Подлинную аварийную печать в программах, написанных на фортране, можно выполнить только используя средства ОС (см. ниже), но сервисные подпрограммы OVERFL и DVCHK позволяют осуществить печать, подобную аварийной. Указанные подпрограммы присваивают переменной, являющейся аргументом подпрограммы, целое значение 1 (или 3) при возникновении ошибок переполнения (или исчезновения) порядка (OVERFL) или при делении на нуль (DVCHK); в противном случае индикативной переменной присваивается значение 2.

Пример:

```

. . . . .
13 IF (.NOT. G) S(I)= A(I)*V(I)-T/A(I)+1.4
C   НАЧАЛО АВАРИЙНОЙ ПЕЧАТИ 3
    CALL OVERFL (KO)
    CALL DVCHK (KD)
    IF (KO.EQ.2 .AND. KD.EQ.2) GO TO 22
        печать
    STOP
C   КОНЕЦ АВАРИЙНОЙ ПЕЧАТИ 3
22 . . . . .
```

Если при выполнении условного оператора с меткой 13 возникнет ошибка (переполнение, исчезновение порядка или деление на нуль), то осуществится заданная печать и выполнение программы закончится.

Печать может быть осуществлена как обычным способом (по WRITE), так и с помощью сервисной подпрограммы DUMP, которая осуществляет печать значений переменных в необходимой форме и прекращает вычислительный процесс. Например, операторы печати и STOP в предыдущем примере могут быть заменены на следующий оператор:

CALL DUMP (G, G, 2, A(1), A(50), 5, V, V, 0, I, I, 4)

Числа 2, 5, 4 задают печать соответственно логических (с длиной 4), вещественных (длина 4) и целых (длина 4 байта) величин. Число нуль задает печать в шестнадцатеричном виде; такая печать может быть задана для переменных любого типа. (Подпрограмма DUMP печатает и соответствующие адреса ячеек памяти, где располагаются переменные).

Ниже приводятся все числа, управляющие видом печатаемых значений:

- 0 — шестнадцатеричный;
- 1 и 2 — логический (1 и 4 байта);
- 3 и 4 — целый (2 и 4 байта);
- 5 и 6 — вещественный (4 и 8 байтов);
- 7 и 8 — комплексный (8 и 16 байтов);
- 9 — литерный.

Для вывода на печать можно воспользоваться и оператором NAMELIST, с помощью которого по оператору WRITE осуществляется вывод значений переменных вместе с их именами по 8 значений в строчке печати.

Отличием печати по индикаторам ошибок от подлинной аварийной печати является то, что она производится не в момент возникновения ошибки, а только после обращения из программы к индикативным подпрограммам, после того, как программно обнаружится наличие ошибки в выполненном ранее операторе; т. е. момент аварийной выдачи определяет, в конечном счете, не система, а программист.

Вставлять обращения к индикативным подпрограммам OVERFL и DVCHK после каждого оператора, где возможна ошибка, разумеется, практически непригодно и поэтому приходится тщательно продумывать места расположения операторов, обращающихся к этим подпрограммам. Если запоздать с обнаружением ошибки и с аварийной печатью, то по выданным результатам будет уже значительно труднее обнаружить ошибку.

З а м е ч а н и я. 1. Индикатор ошибки сохраняет в системе значение, соответствующее ошибке (1 или 3), до тех пор пока не произойдет обращение к индикативной подпрограмме, после чего он принимает значение 2.

2. При возникновении ошибки одного из перечисленных выше типов система осуществляет выдачу диагностической печати и продолжает выполнение программы с неправильными (для переполнения порядка и деления на нуль) результатами. Для ошибок, отличных от указанных выше, система производит диагностическую печать и прекращает выполнение программы (может быть, с осуществлением системного дампа — см. ниже).

Алгол. Аварийная печать в программах, написанных на алголе ЕС, может быть осуществлена только с помощью средств ОС (см. ниже).

Общее замечание. Для того чтобы можно было легко включать и выключать аварийную печать, ее следует задавать по условию, например, следующим образом.

Для PL/1: ON ERROR BEGIN; IF D1 THEN PUT DATA; END;
или
IF D1 THEN ON ERROR PUT DATA;
ELSE ON ERROR;

Пустой оператор в ON-операторе после ELSE введен для отмены возможного оператора ON во внешнем блоке.

Для фортрана: IF(D1) CALL DUMP (A(1), A(50), 5, V, V, 0)
или
NAMELIST /D01/A, V
IF (D1) WRITE (06, D01)

Эти операторы должны замещать оператор печати в приведенном выше примере для фортрана.

Если при вводе D1 присвоить исходное значение соответствующее истине, то аварийная печать будет включена. Для отключения аварийной печати нужно на внешнем носителе (на перфокарте) задать значение ложь, т. е.: '0'B или .FALSE. или 'FALSE'.

Для условного включения аварийной печати можно использовать и препроцессорные средства (см. П.4). Тогда вместо предыдущего оператора PL/1 нужно задать:

```
% DECLARE DP1 FIXED;  
% DP1 = 1;  
% IF DP1  $\neg$  = 1 % THEN % GO TO M1;  
ON ERROR PUT DATA;  
% M1: ;
```

Если препроцессорной переменной DP1 вместо единицы присвоить другое значение, то ON-оператор в транслируемый текст программы не попадает. Другой вариант использования препроцессорных средств приводится ниже (см. 2.1.2, Б).

Препроцессорные средства могут быть использованы для вставки и устранения операторов, реализующих аварийную печать, и в программах, написанных на фортране, хотя при этом имеются и некоторые ограничения (см. П. 7). В приведенном ранее примере для фортрана условный препроцессорный оператор `%IF` нужно поставить перед `CALL OVERFL (KO)`, а пустой оператор — после `STOP`.

ОС. Аварийная печать может быть произведена в форме системного дампа. При дампе, осуществляемом в случае возникновения ошибки в выполняемой программе (составленной на одном из допустимых в ОС языках), на печать выдается в шестнадцатеричном и в символьном виде содержимое рабочей области программы, то есть вид всех ячеек памяти, отведенных системой для выполнения программы (команды, переменные, константы, а иногда и незанятые ячейки). Кроме того, выдается и некоторая системная информация, которая помогает выявить причину аварии в программе для наиболее сложных случаев ошибок. После осуществления дампа выполнение программы прекращается.

Разбор системного дампа требует специальных знаний и используется, в основном, для отладки программ, написанных на языке ассемблера (может быть, и частично), а при работе на алгоритмических языках он является самым последним средством, к которому приходится иногда прибегать, когда другие средства не дают результата, и если имеются серьезные подозрения на ошибку в операционной системе.

Для получения системного дампа необходимо наличие среди управляющих предложений задания директивы (управляющего предложения языка управления заданиями) вида

```
//GO.SYSUDUMP DD SYSOUT=A
```

или

```
//GO.SYSABEND DD SYSOUT=A
```

В первом случае на печать выдается рабочая область программы в момент возникновения ошибки, а во втором случае, кроме того, распечатывается и ядро системы (резидент ОС).

В PL/1 дамп рабочей области программы можно получить также выполнив оператор вида:

```
CALL IHEDUMP (n);
```

и задав описание

```
DECLARE IHEDUMP ENTRY (FIXED BINARY (31, 0));
```

В начале такого дампа печатается целое десятичное число *n*, используемое для идентификации печати при ее разборе. Среди директив выполняемого задания должна находиться директива `DD` вида:

```
//GO.PL1DUMP DD SYSOUT=A
```


Обращение к программе IHEDUMP можно поставить в ON-операторе для ситуации ERROR вместо PUT DATA (см. выше 2.1.1).

З а м е ч а н и е. Ввиду того, что под рабочую память программы система может отвести более сотни килобайт памяти ЭВМ, непосредственная распечатка ее обычно является нерациональной. Часто для выдачи дампа вместо АЦПУ используют магнитную ленту, которую затем выборочно распечатывают по специальной составленной заранее программе.

2.1.2. Печать в узлах. Печать промежуточных значений переменных, интересующих программиста, часто необходимо производить в выбранных им местах (*узлах*) программы. Иногда такой способ называют печатью в режиме блокировки, так как выполнение программы временно приостанавливается — «блокируется», — а затем, после печати, продолжается дальше.

Печать в узлах позволяет программисту получать значения переменных в произвольные моменты работы программы, а не только после возникновения аварийной ситуации в программе. Необходимость печати в узлах следует из того, что к аварии (и, следовательно, к аварийной печати) приводят лишь некоторые, достаточно грубые ошибки в программе, а для локализации других ошибок требуется выдавать на печать значения переменных в тех местах программы, которые предполагаются наиболее близко расположенными к присутствующим в программе ошибкам.

Печать в узлах может быть осуществлена как обычными средствами языка, так и с помощью специальных средств, введенных в алгоритмические языки ЕС.

А. Средства языка.

В нужное место программы непосредственно вставляются операторы языка для печати значений интересующих программиста переменных или оператор обращения к процедуре, в которой осуществляется такая печать (например, с заданием в качестве аргументов имен печатаемых переменных).

Так как отладочная печать в ряде случаев становится не нужной и мешающей, то вместо того, чтобы то вставлять, то удалять соответствующие операторы, удобнее печать осуществлять по условию. В качестве условия берется значение некоторой специальной переменной (ср. с общим замечанием к аварийной печати). Например, оператор, осуществляющий печать в узле, может иметь вид (для PL/I, фортрана, алгола ЕС):

IF #D THEN печать;

или

IF (QD) печать

или

'IF' DOD 'THEN' печать

Одна и та же управляющая переменная может обслуживать несколько печатей.

Б. Специальные средства.

PL/1. В PL/1 для осуществления печати в узлах могут быть использованы исключительные ситуации и ON-оператор, например, следующим образом. Оператору, перед выполнением которого требуется выполнить отладочную печать, приписывается метка (если оператор не помеченный), и она указывается в префиксе CHECK, помещаемом перед блоком, который подвергается отладке. Необходимые операторы печати помещаются в ON-операторы, в которых вслед за ключевым словом CHECK в скобках задаются упомянутые метки.

Рассмотрим, например, следующий участок программы:

```
BEGIN ;
. . . . . }
GO TO M;
X = X + 1;
. . . . .
M: IF Y < 0 THEN CALL P;
. . . . .
```

Для того, чтобы осуществить отладочную печать перед выполнением приведенных в примере операторов, необходимо добавить метки к операторам перехода и присваивания (см. пример ниже); такие метки удобно располагать на отдельных строчках (перфокартах) программы. «Буква» \square использована для удобства программистов, чтобы указать на то, что метки $\square G$ и $\square R$ являются вспомогательными в данной программе и введены специально для целей отладки. Кроме того, нужно приписать префикс к BEGIN, а в начале блока задать ON-операторы с требуемой печатью. Тогда мы получим программу следующего вида:

```
(CHECK( $\square G$ ,  $\square R$ , M)):
BEGIN ;
ON CHECK ( $\square G$ , M) PUT DATA (Z) ;
ON CHECK ( $\square R$ ) PUT DATA (X, Y) ;
. . . . .
 $\square G$ : GO TO M;
. . . . .
 $\square R$ :
X = X + 1;
M: IF Y < 0 THEN CALL P;
. . . . .
```

Чтобы выключить отладочную печать, достаточно перед новой трансляцией убрать некоторые метки из префикса CHECK, который

поэтому удобно располагать перед меткой внешней (или главной) процедуры и записывать на нескольких строчках, например, следующим образом:

```
(CHECK (QG));
(CHECK (QR));
(CHECK (M));
DESIGN: PROCEDURE (A, B, C)
    . . . . .
    BEGIN;
    . . . . .
```

Для управления печатью в узлах можно также воспользоваться и условной печатью в ON-операторе, например:

```
ON CHECK (M) BEGIN; IF Q2 THEN PUT DATA (X, Y); END;
```

Отключение по условию самого ON-оператора:

```
IF Q2 THEN ON CHECK (M) PUT DATA (X, Y);
```

привело бы к печати системной диагностики на возникновение ситуации CHECK (в данном случае печатались бы только рассматриваемые метки).

Управление печатью в узлах удобно осуществлять с помощью препроцессорных средств, обходя в нужных случаях трансляцию отладочных операторов по условию. Например (ср. с общим замечанием в 2.1.1):

```
% DECLARE @P2 FIXED ;
% @P2=1 /* ИЛИ @P2=0; */;
% IF @P2 7=1 % THEN % GO TO @M2;
    (CHECK (QG)) :
    (CHECK (QR)) :
% @M2: ;
    BEGIN;
% IF @P2=1 % THEN
% DO;
    ON CHECK (QG)
        BEGIN; IF ... THEN печать END ;
    . . . . .
    ON CHECK (QR)
        BEGIN; IF ... THEN печать END ;
% END/* IF */;
    . . . . .
QG: GO TO M;
    . . . . .
```

В примере показаны два разных способа использования препроцессорных средств для обхода некоторых участков транслируемой программы (префиксов и операторов ON), управляемых в обоих случаях препроцессорной переменной (в примере — @P2). Поскольку для управления отладочной печатью с помощью препроцессорной переменной требуется каждый раз перетранслировать программу, препроцессорные средства следует применять для обхода целых групп операторов, реализующих отладочную печать; для более динамического включения и выключения печати в узлах удобнее использовать обычные средства языка (см. ON-операторы в приведенном выше примере).

Фортран. Печать в узлах в фортране ЕС может быть осуществлена с помощью операторов так называемого отладочного пакета. Для этого в отладочный пакет DEBUG помещается оператор AT, содержащий метку того оператора, перед выполнением которого необходимо произвести отладочную печать. Вслед за оператором AT, располагаются операторы, реализующие необходимую печать. В качестве таких операторов, помимо обычных операторов печати, в отладочном пакете может быть использован и оператор DISPLAY, осуществляющий печать значений переменных, имена которых заданы (перечислены через запятую) в операторе вслед за ключевым словом.

Пример.

```

REAL X, Y, Z
19  GO TO 24
28  X = X + 1
35  IF (Y .LT. 10) CALL P
    . . . . .
    DEBUG
      AT 19
        DISPLAY Z
      AT 28
        IF (QD1 .EQ. 1) WRITE (6, 900) Z
900  FORMAT ('Z', 900/(E15.8))
      AT 35
        IF (QD2 .NE. 1) GO TO 990
        DISPLAY X, Y
990  CONTINUE
    END

```

Во время выполнения программы перед операторами с метками 19 и 28 будет производиться печать Z, а перед оператором IF — печать X и Y; осуществление печати для операторов 28 и 35 управляется значениями переменных QD1 и QD2 (см. общее замечание к 2.1.1).

Для вставки и устранения операторов отладочного пакета (в особенности для программ, хранящихся во внешней памяти) можно воспользоваться препроцессорными средствами (см. П.4). В нашем примере перед первым оператором программы следовало бы добавить операторы (ср. с общим замечанием к 2.1.1):

```
% DECLARE #P FIXED;  
% #P = 1 /* ИЛИ % #P = 0 */;
```

а перед и после группы устранимых операторов отладочного пакета добавить соответственно операторы

```
% IF #P = 0 % THEN % GO TO #M;
```

и

```
% #M: ;
```

Конечно, для одноразовой вставки или устранения отладочных операторов можно просто воспользоваться программой UPDATE (см. гл. 3).

Общее замечание. Для того, чтобы можно было использовать в PL/1 и фортране приведенные выше способы осуществления отладочной печати в узлах, операторы, перед которыми необходимо выполнить печать, должны быть помеченными. Поэтому расположение узлов в программе следует намечать заранее по тексту разработанного алгоритма, а при написании программы снабжать выбранные «узловые» операторы метками.

Алгол. В алголе ЕС какие-либо специальные средства, помогающие осуществить печать в узлах, отсутствуют, и для реализации печати приходится прибегать к обычным средствам языка, вставляя условную печать в нужные места программы.

2.1.3. Слежение. Одной из форм контроля за ходом выполнения отлаживаемой программы является *слежение* за использованием в программе выбранных переменных или за выполнением некоторых, интересующих программиста операторов. Первое слежение назовем *арифметическим*, второе — *логическим* (хотя оба используются, в основном, при проверке логики программы).

А. Арифметическое слежение.

При арифметическом слежении обычно осуществляется печать значений выбранной программистом переменной (а, может быть, и других переменных) в те моменты, когда ей присваивается какое-либо значение.

PL/1. В PL/1 арифметическое слежение осуществляется с помощью ситуации CHECK. В префиксе CHECK, приписываемом к какому-либо блоку (простому или процедурному), задаются имена тех переменных в блоке, за изменениями которых требуется проследить.

В тот момент, когда указанной переменной присваивается новое значение, производится печать имени переменной и ее нового значения; печать каждый раз производится с новой строчки. Для случая, когда присваивание производится элементу массива, печатаются значения всего массива, обычно, по 5 значений в строчке АЦПУ; это же относится и к скалярам, входящим в массивы структур и приобретающим вследствие этого атрибут размерности (для скаляров, входящих в «скалярные» структуры печатается одно значение). Если задан ON-оператор, управляемый соответствующей ситуацией CHECK (содержащей в скобках имя рассматриваемой переменной), то производится указанная в нем печать (или какие-либо другие действия), а имя изменяемой переменной и ее значение не печатаются, и об этом нужно позаботиться самому программисту; перевод на новую строчку АЦПУ для каждой печати из ON-оператора автоматически не производится.

Отметим, что ситуация CHECK возникает и печать производится также в том случае, если при присваивании или вводе новое значение переменной совпадает со старым, то есть фактически не меняется.

Пример.

```
(CHECK (X, Y, Z)):
G: BEGIN ; DECLARE X, Y, Z (20);
    ON CHECK (Y) PUT DATA (Y, T, A);
    . . . . .
    GET LIST (X, Y);
    D = X*2 + Y;
    Z (I) = D;
    . . . . .
END G;
```

После выполнения оператора ввода будут напечатаны имя и значение X, а также имена и значения Y, T, A; каждый раз при присваивании Z (I) печатаются все значения массива Z, с их именами и индексами. После любого оператора блока G, в котором происходит изменение X, Y или Z, также будет производиться указанная отладочная печать.

Фортран. В фортране ЕС арифметическое слежение может быть осуществлено с помощью отладочного оператора DEBUG, устанавливающего режим INIT.

Пример.

```
REAL X, Y, Z (20)
. . . . .
READ (5, 15) X, Y
15      FORMAT ...
      D = X*2 + Y
```

```

      Z(I)=D
      . . . . .
DEBUG INIT (X, Y, Z)
END

```

После выполнения оператора ввода будет производиться печать значений X и Y, а после присваивания Z(I) печатается его значение для конкретного I; при этом печатаются также имена соответствующих переменных (простых или с индексами), но вместо индексов печатается порядковый номер элемента в массиве, располагаемого в памяти по столбцам.

Алгол. В алголе ЕС средств для осуществления арифметического слежения нет.

Б. Логическое слежение.

Логическое слежение позволяет узнать последовательность (и факт) выполнения операторов и процедур в программе.

PL/I. В PL/I логическое слежение осуществляется с помощью префикса (ситуации) CHECK, в котором перечисляются метки операторов или имена процедур, за выполнением которых необходимо проследить. При выполнении таких операторов будет производиться печать соответствующих меток или имен процедур. Пример для меток операторов можно найти в 2.1.2, А. Префикс относится только к тем блокам и процедурам, которые находятся внутри префиксированного блока.

Фортран. В фортране для слежения за выполнением подпрограмм, необходимо в каждую такую подпрограмму вставить оператор DEBUG с режимом SUBTRACE. При входе в такую подпрограмму и выходе из нее печатаются соответственно сообщения вида:

SUBTRACE имя-подпрограммы

и

SUBTRACE *RETURN*

Удобных средств для слежения за выполнением некоторого помеченного оператора в фортране нет, но такое слежение можно осуществить с помощью режима TRACE в операторе DEBUG и операторов TRACE ON и TRACE OFF, располагаемых в двух соседних операторах AT.

```

Пример.
      REAL X, Y, Z
19      . . . GO TO 35 . . .
      . . . . .
28      X = X + 1
29      . . . . .
      . . . . .

```

```

35      IF (Y .LT. 0) CALL P
        DEBUG TRACE
          AT 19
            TRACE ON
          AT 35
            TRACE OFF
          AT 28
            IF (OD .NE. 1) GO TO 902
              TRACE ON
902      CONTINUE
          AT 29
            TRACE OFF
        . . . . .
      END

```

При выполнении операторов с метками 19 и 28 будет производиться печать вида (в последнем случае по условию):

TRACE 19 и TRACE 28

Для отладочных средств фортрана более естественным является слежение, производимое не за отдельным оператором, а за всеми помеченными операторами на некотором участке (см. ниже п.2.1.4, Б).

Для того, чтобы проследить за обращением к подпрограмме Р, в нее перед оператором END необходимо вставить следующий оператор:

DEBUG SUBTRACE

Об использовании препроцессорных средств для отключения отладочной печати см. 2.1.1 и 2.1.2, Б.

Алгол. В алголе ЕС нет специальных средств, предназначенных для логического слежения, но в случае, если требуется проследить только за одним оператором в программе, можно воспользоваться средствами для логической прокрутки (см. ниже 2.1.4, Б).

Общие замечания. Каждая метка или значение при слежении печатается с новой строчки (или через строчку), что может привести к слишком большому объему печати, если проверяемые операторы находятся в цикле. О способах сокращения отладочной печати см. 2.2.3÷2.2.4.

2.1.4. Прокрутка. Прокрутка предназначена для получения наиболее полной информации о вычислениях, производимых на некотором участке программы (*арифметическая прокрутка*) или о последовательности выполнения всех операторов на этом участке (*логическая прокрутка*).

А. Арифметическая прокрутка.

При арифметической прокрутке для программы, написанной на алгоритмическом языке, обычно печатаются результаты выполнения

каждого оператора присваивания на заданном участке программы (для программ на языке ассемблера — результаты каждой команды).

PL/1. В PL/1 нет специальных средств для реализации арифметической прокрутки; на небольших участках прокрутку можно осуществить с помощью префикса CHECK, в котором указаны имена всех переменных, фигурирующих в левых частях операторов присваивания (см. выше 2.1.3, А).

Фортран. В фортране ЕС арифметическая прокрутка осуществляется с помощью отладочного режима INIT, если для него не задан список имен переменных. В этом случае слежение будет производиться за всеми переменными подпрограммы, встречающимися в левых частях операторов присваивания (см. выше 2.1.3, А).

Алгол. В алголе ЕС нет специальных средств для реализации арифметической прокрутки.

Б. Л о г и ч е с к а я п р о к р у т к а.

При логической прокрутке обычно печатаются метки или номера всех операторов, выполняемых на заданном участке программы.

PL/1. В PL/1 нет специальных средств для реализации логической прокрутки.

Фортран. В фортране ЕС логическая прокрутка осуществляется с помощью отладочного режима TRACE и операторов TRACE ON и TRACE OFF, задаваемых после двух операторов AT, соответственно содержащих метки начала и конца прокрутки. Данный режим использовался ранее (см. 2.1.3, Б) для логического слежения. Для печати меток всех выполняемых операторов на участке от метки 28 до 35 (исключая последнюю), достаточно задать:

DEBUG TRACE

AT 28

TRACE ON

AT 35

TRACE OFF

Включение в прокрутку операторов, содержащихся в отдельных подпрограммах, производится заданием аналогичных отладочных операторов в каждой такой подпрограмме.

Алгол. В алголе ЕС логическая прокрутка выполняется при задании опции транслятора TEST и параметров TRACE, TRBEG=*m* или TREND=*n* для шага выполнения программы (для шага GO). Параметр TRACE задает режим прокрутки для всей программы, а пара TRBEG и TREND задает номера (*m* и *n*) начального и конечного операторов в программе, которые должны быть подвергнуты прокрутке; *m* может быть и больше *n*. Например, по директиве вида

//имя EXEC ALGOFCLG,PARM.ALGOL='TEST',

// PARM.GO='TRBEG=14,TREND=40'

для транслируемой и выполняемой затем программы задается логическая прокрутка, начиная с оператора 14 и кончая оператором с номером 40. Задаваемый номер оператора определяется по распечатке программы, выдаваемой транслятором; он соответствует количеству точек с запятой, предшествующих данному оператору в тексте программы (начиная с нуля). Если не задать TREND, то прокрутка будет выполняться до конца программы. Номера выполняемых операторов печатаются по 13 чисел в строчке АЦПУ.

2.1.5. Другие средства.

Контроль индексов. Во всех рассматриваемых трансляторах ОС ЕС может быть включен отладочный режим, предназначенный для обнаружения случая выхода значений индексов переменной за границы, указанные при описании массива. Нужно отметить, что на контроль индексов, установленный для всей программы или большого модуля будет тратиться много машинного времени.

PL/1. В PL/1 контроль индексов задается префиксом SUBSCRIPTRANGE, приписываемом к любому оператору, блоку или процедуре. Диагностическая печать (выдаваемая, если отсутствует ON-оператор) содержит имя ситуации и номер ошибочного оператора.

Фортран. В фортране ЕС контроль за превышением значения индекса задается режимом SUBCHK в операторе DEBUG; вслед за словом SUBCHK в скобках указывается список контролируемых имен массивов. Если имена массивов не указаны, то контролируются переменные с индексами для всех массивов, описанных в данной программной единице. При обнаружении выхода значения индекса за границы, печатается сообщение вида:

SUBCHK имя-массива (порядковый-номер-элемента-массива)

Алгол. В алголе ЕС контроль за значениями индексов задается с помощью параметра транслятора TEST (см. выше пример). В случае ошибки на печать выдается диагностическое сообщение и номер ошибочного оператора.

Печать внешней памяти. Наряду с печатью значений переменных, находящихся в оперативной памяти, часто приходится распечатывать и значения переменных, расположенных во внешней памяти. В этом случае используются служебные программы (утилиты), предназначенные для работы с внешними носителями. В ОС ЕС ЭВМ к служебным программам [14] можно обращаться непосредственно из программ, написанных на алгоритмических языках (для PL/1 и Фортрана).

2.1.6. Классификация средств. В таблицу 1 сведены отладочные средства ОС ЕС ЭВМ, упомянутые выше в п. п. 2.1.1÷2.1.5. Изложенные средства локализации различаются и по принципам своей работы, и по типам обрабатываемых или выводимых данных, и по

Отладочные средства ОС ЕС ЭВМ

	PL/I	Фортран	Алгол	ОС
Аварийная печать (п. 2.1.1)	ON ERROR, PUT DATA, CALL INEDUMP в PLI- DUMP DD, INITIAL	CALL OVERFL, CALL DVCHK, NAMELIST, CALL DUMP, /нач. значения/	DUMP	SYSUDUMP DD SYSABEND DD
Печать в узлах (п. 2.1.2, А и 2.1.2, Б)	Операторы печати, CHECK (метки), ON CHECK (...)	Операторы печати, AT, DISPLAY	Операторы печати	—
Слежение арифметическое (п. 2.1.3, А)	CHECK (имена)	DEBUG INIT (имена)	—	—
Слежение логическое (п. 2.1.3, Б)	CHECK (метки) CHECK (входы)	DEBUG TRACE, AT, TRACE ON, TRACE OFF, DEBUG SUBTRACE	—	—
Прокрутка арифметическая (п. 2.1.4, А)	—	DEBUG INIT	—	—
Прокрутка логическая (п. 2.1.4, Б)	—	DEBUG TRACE, AT, TRACE ON, TRACE OFF	TEST, TRACE, TRBEG, TREND	—
Контроль индексов (п. 2.1.5)	SUBSCRIPTRANGE	DEBUG SUBCHK	TEST	—
Распечатка внешней памяти (п. 2.1.5)	см. ОС	см. ОС	—	Служебные программы

способам доступа к ним (в зависимости от их принадлежности к различным частям ОС).

По принципам работы средства локализации были разделены на 4 типа: аварийная печать, печать в узлах, слежение, прокрутка. Характер задаваемой программистом информации при обращении к какому-либо из средств зависит от его типа.

Аварийная печать осуществляется один раз при работе отлаживаемой программы, в момент возникновения аварийной ситуации в программе, препятствующей нормальному ее выполнению. Тем самым, конкретное место включения в работу аварийной печати определяется автоматически без использования информации от программиста, который должен только определить список выдаваемых на печать переменных.

Печать в узлах включается в работу в выбранных программистом местах программы; после осуществления печати значений заданных переменных продолжается выполнение отлаживаемой программы.

Слежение производится или по всей программе, или на заданном программистом участке. Причем слежение может осуществляться как за переменными (арифметическое слежение), так и за операторами (логическое слежение). Если обнаруживается, что происходит присваивание заданной переменной или выполнение оператора с заданной меткой, то производится печать имени переменной или метки и выполнение программы продолжается. Отличием от печати в узлах является то, что место печати может точно и не определяться программистом (для арифметического слежения); отличается также и содержание печати.

Прокрутка производится на заданных участках программы, и после выполнения каждого оператора заданного типа (например, присваивания или помеченного) происходит отладочная печать.

Схематически изложенные принципы работы рассматриваемых средств изображены на рис. 3. Сплошная тонкая линия — путь выполнения программы, пунктирная — отсутствие выполнения программы, жирная — сплошное осуществление отладочных действий в ходе работы программы. Точки изображают места включения отладочных средств и выполнения отладочной печати; места, непосредственно указываемые программистом, изображены сплошными точками, а светлые точки указывают места в программе, определяемые с помощью системных средств.

По типам печатаемых значений (числовые и текстовые или меточные) средства разделяются на арифметические и логические.

Средства могут быть общесистемными или относиться к одному из трансляторов, причем в последнем случае различаются средства, реализуемые обычными операторами языка или с помощью специальных его возможностей (в PL/1 к таким отнесены ON-ситуации).

Отметим различный характер задания информации для специальных отладочных средств в рассматриваемых здесь трансляторах. В алголе необходимая информация задается вне программы, а именно, в директиве EXEC, включающей в работу транслятор. В фортране отладочные операторы задаются в программе, но они составляют особую группу и собраны в конце каждого программного модуля. В PL/1 отладочные средства могут располагаться внутри программы в произвольных местах,

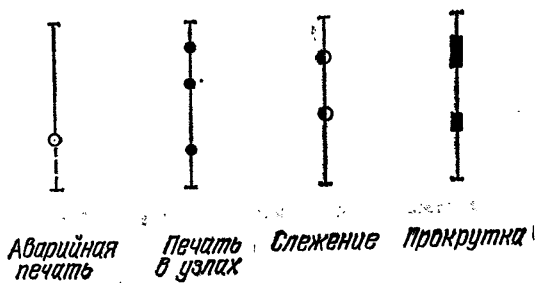


Рис. 3. Принципы действия средств локализации.

Ниже дана классификация средств локализации, рассмотренных выше в этой главе.

Типы средств локализации:

1. Языковые.
 - 1.1. Обычные.
 - 1.2. Специальные.
2. Системные.
- (3. Пакетные)

Средства локализации:

1. Аварийная печать (арифметическая).
 - 1.1. Специальные средства языка.
 - 1.2. Системные средства.
2. Печать в узлах (арифметическая).
 - 2.1. Обычные средства языка.
 - 2.2. Специальные средства языка.
3. Слежение (специальные средства).
 - 3.1. Арифметическое.
 - 3.2. Логическое.
4. Прокрутка (специальные средства).
 - 4.1. Арифметическая.
 - 4.2. Логическая.

2.2. Использование средств

Здесь дается сравнение и некоторые рекомендации по использованию средств локализации, изложенных выше.

2.2.1. Сравнение средств. Сравним возможности изложенных средств и дадим некоторые советы, касающиеся их использования.

Когда встает вопрос о выборе того или иного средства локализации в ходе отладки, то начинающие программисты обычно выбирают прокрутку всей программы (для фортрана и алгола ЕС) или полную аварийную печать (для PL/I). Критерием для такого выбора является то, что эти средства требуют от программиста задания минимальной информации, а на печать выдают максимальное количество результатов, с помощью которых, как кажется, можно будет найти любую ошибку. Вследствие такого стремления к экономии своего мышления, программисту обычно выдается на печать такое количество результатов, что для их полного разбора потребовалось бы несколько дней. После этого у программиста надолго пропадает охота применять какие-либо отладочные средства, хотя они мало виноваты в том, что их использовали не по назначению и выбрали из других только потому, что они не требуют большой исходной информации и хороших знаний алгоритма отлаживаемой программы.

Сопоставляя изложенные средства, нужно сразу же отметить, что основным, наиболее гибким и эффективным средством, используемым для локализации ошибок, является печать в узлах. Переменные, выводимые на печать, и узлы намечаются программистом, в основном, на стадии алгоритмизации или после ее окончания, но до этапа программирования. Именно на этапе алгоритмизации внимание программиста сосредоточено на логической структуре программы, на взаимосвязи ее блоков, на назначении и взаимодействии переменных. Поэтому ему легче установить, в каких местах и какие переменные нужно напечатать, чтобы по их тестовым результатам можно было легко локализовать ошибку, в каком бы месте программы она ни возникла.

Слежение и, реже, прокрутка (логические и арифметические) используются, в основном, как дополнения к методу печати в узлах, в тех случаях, когда не удастся по отпечатанным результатам выявить ошибку, или для отладки некоторого участка программы, в котором существенно преобладают логические операторы. Причем прокрутка является, обычно, менее предпочтительным средством поскольку выдает слишком много тестовых результатов, в которых приходится долго разбираться, чтобы найти нужные.

Аварийная печать также не может быть использована в качестве основного средства локализации ошибок, так как отсутствие аварии в программе еще не говорит о том, что в ней нет ошибок. Аварийная

печать часто используется вместо печати в узлах теми программистами, которые не намечают заранее плана отладки, а в качестве тестовых данных задают реальные данные и надеются выловить все ошибки в программе по печатаемым окончательным результатам программы. Если выполнение программы не доходит до печати результатов из-за ошибок в ней, то на этот случай и предусматривается аварийная печать. Но для некоторых ошибок (например, для заикливания) аварийной печати может и не быть, и все равно придется прибегать к другим способам локализации ошибок.

Аварийную печать можно использовать на стадии пробной или даже нормальной эксплуатации программы в качестве замены печати в узлах. После отладки программы операторы печати в узлах из программы убираются или выключаются, а затем включается аварийная печать, которая и остается в программе на этапе счета, на случай возникновения аварии в программе, для выявления ее причины разработчиком или по его инструкции — сопроводителем.

О случаях применения системной аварийной печати было сказано ранее (см. 2.1.1).

2.2.2. Идентификация печати. Каждая отладочная печать должна иметь какие-то свойственные только ей внешние признаки, по которым можно было бы легко выделить ее из всей напечатанной информации. Этим признаком может быть и количество печатаемых данных, и формат их печати, и какой-то специальный текст, помещаемый в начало отладочной печати. Например:

ПЕЧАТЬ ИЗ УЗЛА GOL

Здесь GOL — метка оператора, перед выполнением которого произведена печать.

Существенно помогает разбору выдаваемых результатов и использование тех видов операторов печати, при которых наряду с результатами печатаются и имена соответствующих переменных: PUT DATA в PL/1 или DISPLAY и WRITE с NAMELIST в фортране.

2.2.3. Минимизация печати. При использовании отладочной печати необходимо следить за тем, чтобы печатались лишь данные, которые действительно необходимы для локализации ошибок. Нужно особенно тщательно отбирать массивы, выводимые на печать, и стремиться к тому, чтобы минимизировать их количество. Лишняя печать не только замедляет работу машины, но, главное, затрудняет разбор выдачи и поиск необходимых тестовых данных среди напечатанных. Особенно нужно следить за теми средствами печати, которые размещаются во внутренних циклах. В противном случае может оказаться, что объем отладочной печати превысит разумные пределы или ресурсы операционной системы.

При необходимости размещения печати внутри больших по кратности циклов, если уменьшение кратности в данном тесте невозможно, следует предусмотреть отключение печати после того, как цикл выполнится нужное число раз. Отключение отладочной печати, осуществляемой средствами языка, может быть реализовано, например, в зависимости от значений параметра цикла или по счетчику. В PL/I для отключения печати, осуществляемой по ON-оператору вместо отключения операторов печати можно в нужный момент выполнить новый оператор ON с пустым оператором в качестве его тела.

Пример.

```
ON CHECK (OD) PUT DATA (X, Y, Z);
DO I=1 TO N;
  IF=4 THEN ON CHECK (OD);
  . . . . .
END;
```

Если ситуация CHECK не используется, то печать может быть, например, задана так:

```
IF OD=1 & (I<=3 I>N-2) THEN PUT LIST(A, B, C);
```

Печать будет производиться для трех первых и двух последних выполнений цикла при условии, что OD равно единице. Аналогичное управление отладочной печатью внутри цикла может быть осуществлено для фортрана и алгола.

2.2.4. Автоматизация сверки. Одним из средств минимизации печати является автоматизация сверки печатаемых результатов. При этом на печать выдаются только те результаты, которые не совпадают с эталонными; должна печататься и дополнительная информация, каким-то образом идентифицирующая печатаемые результаты (привязывающая их к соответствующим переменным, элементам массивов).

В ОС ЕС ЭВМ нет языковых или системных средств для сверки данных, если не считать служебной программы COMPR, которая не предназначена для целей отладки. Ниже рассмотрены некоторые возможные варианты использования метода автоматической сверки при отладке, которые могут быть реализованы программистом непосредственно или путем создания специальных сервисных программ.

1. Заготовленные эталонные результаты препарируются и вводятся в оперативную или внешнюю память ЭВМ и с ними сверяются в дальнейшем получаемые тестовые результаты. Для арифметических данных может потребоваться задание и требуемой точности сверки, а для данных другого типа — информация, указывающая на поля данных, подвергаемые сверке.

2. В качестве эталонных результатов используются результаты, полученные ранее при счете по отлаживаемой программе, признанные правильными (или почти правильными) и записанные во внешнюю память (для больших массивов — обычно на магнитную ленту); см. рис. 4, схемы 1 и 2. Этот вариант хорош тем, что не требуется препа- рировать большие объемы данных. Он применяется, обычно, на позд- них стадиях отладки, когда после внесения некоторых изменений или улучшений, требуется проверить не повредили ли эти локальные из- менения работе остальных блоков программы.

Часто оказывается удобнее и сверяемые данные перед выполне- нием сверки предварительно записывать во внешнюю память (см. рис. 4, схемы 3 и 4).

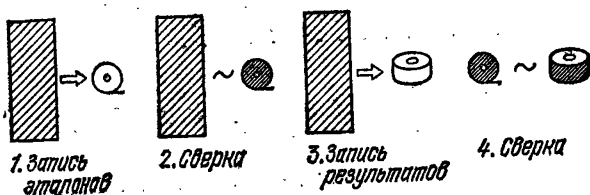


Рис. 4. Автоматизация сверки результатов.

3. Сверку можно применять не только для результатов, печатае- мых в программе, но и для информации, выдаваемой на печать систе- мой, в соответствии с заданными отладочными режимами. В частно- сти, результаты логического слежения или прокрутки, которые при печати занимают много места (так как располагаются через строчку), можно, подвергнув сверке и переработке, преобразовать, например, к следующему виду:

для цикла	для цикла в цикле
*n КРАТНОСТЬ метка	**m КРАТНОСТЬ метка-1
	*n КРАТНОСТЬ метка-2

Здесь метка-1 находится во внешнем, а метка-2 — во внутрен- нем цикле; m и n — конкретные числа (кратности выполненных цик- лов).

Для того, чтобы реализовать такое сжатие информации, нужно направить выводимую информацию не на системную печать, а во внешнюю память, изменив для этого директиву DD с именем SYS- PRINT в шаге GO. Записанная информация обрабатывается затем по специальным программам, которые могут быть разработаны систем- ными или проблемными программистами любой организации.

В фортране имеется возможность направить результаты отла- дочной печати во внешнюю память, задав необходимый ссылочный

номер набора данных в режиме UNIT оператора DEBUG и соответствующую директиву DD, например:

DEBUG UNIT (09)

и

//FT09F001 DD характеристики-набора-данных

Такой режим может быть задан в любой программной единице фортрана.

2.2.5. Включение отладочных средств. Удобство использования отладочных средств зависит от того, насколько просто можно при желании выключить их или производимую ими печать, а затем при необходимости снова включить. Ниже дается сводка способов включения и выключения основных отладочных средств (часть из них упоминалась и ранее).

1. Обращение к отладочным средствам перфорируется на отдельных перфокартах, и для включения необходимого средства перфокарты просто вставляются в нужное место колоды, а для выключения — вынимаются. При хранении программы во внешней памяти используются имеющиеся средства исправления программ (см. ниже гл. 3).

2. Отладочные средства языка включаются по условию, с использованием условных операторов языка. Исходные значения, управляющие включением и выключением средств, вводятся с внешних носителей (обычно с перфокарт) в самом начале работы программы; на перфокартах они располагаются перед прочими исходными данными, чтобы их легко можно было найти.

В PL/I управляющие значения удобно вводить по оператору GET DATA, а в фортране — с помощью NAMELIST, с тем чтобы можно было не заботиться о порядке расположения перфокарт, содержащих эти значения.

3. Для включения отладочной печати или режима могут применяться и препроцессорные средства (см. П.4).

Препроцессорный оператор:

% IF @G1=1 % THEN % DO; печать-или-режим % END;

вставляет печать или режим в текст программы только в том случае, если препроцессорная переменная @G1, управляющая печатью, имеет значение равное единице; описание этой переменной и присваивание ей необходимого значения удобно производить перед началом текста программного модуля (см. общее замечание в 2.1.1 и пример в 2.1.2, Б). В отличие от предыдущего способа (п.2) команды, реализующие отладочную печать, при @G1≠1 не будут содержаться в выполняемой программе, и не будет тратиться время на проверку необходимости печати. Но для включения и выключения печати необ-

ходимо каждый раз производить перетрансляцию (что не требуется для предыдущего способа).

4. Вместо того, чтобы непосредственно использовать операторы печати тестовых данных, можно обращаться к некоторой составленной заранее процедуре, производящей отладочную печать. Проверка необходимости выполнения отладочной печати программируется при этом только внутри такой процедуры, а не при каждом обращении к ней из узлов программы. Трудности реализации этого способа могут быть вызваны только тем, что процедура печати должна иметь несколько входов, различающихся количеством и типом параметров; в PL/1 для обхода этой трудности удобно воспользоваться атрибутом общего входа — **GENERIC**.

З а м е ч а н и е. Операторы отладочной печати (может быть, и в выключенном состоянии) обычно остаются в программе до окончания отладки блока или программы, после чего они устраняются, а блоки перетранслируются и еще раз подвергаются тестированию для выявления внесенных при этом ошибок. Некоторые основные отладочные печати должны быть оставлены в программе в выключенном состоянии для использования при доработках в ходе эксплуатации программы. Устранение отладочных печатей, оставляемых в программе на поздних этапах разработки, очень удобно производить с помощью препроцессорных средств, спланированных, разумеется, заранее.

2.3. Методика локализации

Здесь приводятся основные принципы и некоторые приемы локализации ошибок по тестовым данным, полученным с помощью средств, изложенных ранее.

2.3.1. От симптома к ошибке. Пропуск теста на машине (как и выполнение обычной программы) может окончиться одним из следующих вариантов:

- 1) авария в программе;
- 2) заикливание;
- 3) нормальное окончание с выдачей результатов (может быть, и правильных).

Если в тесте не предусмотрена печать промежуточных результатов, то для первых двух вариантов окончания всегда, а для третьего варианта — при получении неверных результатов, встает задача определения места или участка программы, где находится ошибка. Для первых двух вариантов кажется, что место ошибки уже определено, поскольку в первом варианте им является оператор, где произошла авария, а во втором — оператор, организующий повторение цикла. Но на самом деле может оказаться, что авария и заикливание

ние это лишь внешнее проявление — симптом ошибки, которая находится совсем в другом месте программы. Локализация в том, собственно, и состоит, что по симптому ошибки определяется истинная причина и действительное место ошибки.

Место проявления ошибки позволяет лишь определить с некоторой вероятностью последний оператор того участка программы где содержится ошибка; в частности, для третьего варианта ошибочным оператором может оказаться сам оператор печати результатов. Первым оператором подозреваемого участка программы является первый оператор программы. Таким образом, определяется участок программы, содержащий ошибку; в худших случаях им может оказаться вся программа.

Мы рассматривали случай, когда все результаты программы выводятся в конце ее работы и отсутствует какая-либо отладочная печать. Иначе участком, содержащим ошибку, скорее всего, будет являться часть программы между двумя операторами печати (в частности, операторами печати в узлах), в одном из которых получены все правильные результаты, а в соседнем есть и неверные. Если участок оказался слишком большим или диагностика ошибки представляется слишком сложным делом, необходимо добавить новые узлы отладочной печати внутри участка. Вид и место отладочной печати намечаются по блок-схеме (обобщенному алгоритму) программы с учетом полученных ранее результатов. В некоторых случаях участок просто делится пополам и узел вставляется в его середину.

Для небольшого участка, чтобы найти в нем конкретное место ошибки, обычно оказывается возможным не прибегать к получению новых тестовых результатов, а применить способ обратного отслеживания (см. ниже 2.3.2).

З а м е ч а н и е. В простейшем случае участком является некоторый фрагмент текста программы, выполняемый последовательно и состоящий из нескольких операторов или блоков. В реальной программе, содержащей циклы и условия, под участком следует понимать фрагмент процесса, определяемого программой, то есть как бы текста, развернутого во времени выполнения программы. В этом случае авария может быть вызвана и ошибкой в операторе (блоке), расположенном далее по тексту программы.

2.3.2. Обратное отслеживание. Если предположить, что программист имеет в своем распоряжении все необходимые для проверки подозреваемого участка тестовые и эталонные промежуточные результаты, то для обнаружения места ошибки в программе достаточно их просто сверить между собой и установить, к какому оператору в программе относится первый несовпавший с эталонным тестовый результат. Это относится как к арифметическим, так и к логическим результатам; в качестве последних берется след (трасса) программы.

Реально, исходя из требования минимизации печатаемых результатов, программист имеет в своем распоряжении не все промежуточные результаты, а только основные. Рассмотрим на примере методику обнаружения неверного оператора в программе для этого случая.

Пусть имеется участок программы, на котором вычисляются переменные A, B, C, D, функционально зависящие от указанных ниже переменных:

```
.....  
A=R(X, Y);  
B=Q(X, Y, A);  
C=R(X, Y, A);  
D=S(X, Y, C);  
.....
```

Имена функций P, Q, R, S можно считать условными обозначениями для указания некоторых функциональных зависимостей или именами реальных процедур-функций.

Предположим, что X и Y были вычислены, отпечатаны и проверены на предыдущем участке программы и оказались верными; значения переменных A, B, C не печатаются, а D получилось неправильным. Можно предположить, что ошибка заключается в вычислении C или в самом операторе, вычисляющем D. Для того чтобы локализовать ошибку, нужно узнать правильно ли вычисляется C. Но правильность C, в свою очередь, зависит от правильности A, и поэтому для того чтобы узнать, где же ошибка, придется проверить правильность вычисления A.

Установление правильности операторов, вычисляющих A и C (и D), может быть произведено путем тщательной проверки или прокрутки операторов, или даже посредством вычисления A и C вручную в соответствии с текстом проверяемых операторов в программе (в случае когда такие вычисления являются достаточно простыми). Если проверка за столом не приводит к успеху, то приходится обратиться к машине, вставив дополнительную отладочную печать величин A и C.

Величина B осталась непроверенной в данном случае, но, может быть, к ней еще придется вернуться в дальнейшем, если она окажется входящей в оператор, подвергаемый проверке описанным выше способом. Это способ иногда называют *обратным отслеживанием*.

Таким образом, применение обратного отслеживания позволяет выдавать на печать не все тестовые результаты, а только те, которые необходимы для установления конкретного участка программы, содержащего ошибку, или те результаты, для получения которых при отслеживании пришлось бы проводить слишком большой объем вы-

числений. Например, если операторы для получения А, В или С являются достаточно сложными, то их результаты следует печатать во время пропуска тестов.

Отметим, что, как правило, для отслеживания бывают необходимы исходные данные (вводимые по операторам ввода), поэтому их следует всегда выдавать на печать при отладке; при большом количестве исходных данных их лучше распечатывать автономно и сверять до пропуска теста.

2.3.3. Природа ошибки. В тех случаях, когда полученные результаты, несмотря на все усилия, никак не удается объяснить и сопоставить с ожидаемыми, эталонными или полученными ранее, полезно вспомнить о том, что в их получении помимо отлаживаемой программы принимали участие машина, операционная система, транслятор, операторы ЭВМ. По статистическим данным лишь 90% ошибок, возникающих в ходе отладки на машине, можно отнести к ошибкам в программах, а остальные приходится на других участников вычислительного процесса, упомянутых выше, и, главным образом, на ЭВМ. Поэтому не стоит приходить в отчаяние, а нужно просто пропустить тот же тест еще раз. Если результат не повторится, то это скорее всего означает, что ранее полученные результаты были испорчены сбоем машины. (Иногда, правда, результаты могут не повторяться и из-за ошибки в программе, когда в ней используется значение переменной, которой перед этим не было присвоено никакого начального значения; значение переменной, а, следовательно и результат выполнения программы, будет зависеть от того, какое содержимое памяти осталось после работы предыдущей программы.) Если же странные результаты повторяются, то для определения природы ошибки в программе ее придется локализовать обычным способом, чтобы максимально сузить место ошибки и убедиться в наличии или отсутствии ошибок программиста в этом месте. В первом случае приходится подозревать операционную систему, в частности, использованный транслятор. Такие подозрения являются не лишними, так как известно, например, что в OS IBM/360 имеется более 1000 ошибок [6]; видимо, столько же их и в ОС ЕС ЭВМ.

В случае, когда подозревается транслятор, следует обратить внимание на те участки программы, в которых применяются редкие, не использованные ранее возможности языка или сложное сочетание, нагромождение обычных конструкций. Имеет смысл упростить реализацию таких участков, отказавшись от сложных и неопробованных возможностей, или посоветоваться с более опытными программистами.

Если имеются подозрения на управляющую программу операционной системы, то следует обратить внимание на объем используемой в рабочей программе памяти (оперативной и внешней), погово-

рять с сопровождающими об особенностях и режиме работы ОС в день пропуска программы (может быть, работа шла лишь на части ресурсов машины или в некотором специфическом режиме).

Отметим, что причиной несовпадения получаемых результатов с эталонными может быть и ошибка при получении самих эталонных результатов. Кроме того, ошибка может содержаться и в документации, и в описании подпрограмм, заимствованных в других организациях.

Поэтому, когда программист заходит в тупик при поиске ошибки в программе, он должен подозревать всех и все, но в первую очередь, конечно, самого себя. В конце концов может оказаться, что ошибка состоит в том, что программист просто неправильно понял описание используемых конструкций языка или забыл о существующих ограничениях. Для проверки своих представлений о возможностях языка, впервые применяемых в программе, полезно еще при составлении программы или во время ее отладки практически опробовать такие возможности на простых «верификационных» тестах.

2.3.4. Некоторые замечания. Отладка программ и, в частном случае, локализация ошибок по своему характеру является творческой работой и сочетает в себе элементы науки и искусства. Программист находит ошибки как путем целенаправленных размышлений, так и с помощью интуиции, основанной на имеющемся опыте.

Поиск ошибок при отладке является трудным, но весьма интересным занятием. Иногда даже кажется, что некоторые программисты терпимо относятся к ошибкам в составленной программе и плохо проверяют ее за столом только потому, что они не хотят в дальнейшем лишаться себя удовольствия поохотиться за ошибками с помощью машины. Они оправдываются тем, что на машине поиск ошибок проходит легче и быстрее, чем за столом. Эксперименты и наблюдения [8, гл. 13] говорят о том, что это впечатление обманчиво. Очень часто одна проверка программы или ее блока позволяет найти за столом столько ошибок, сколько не удастся найти и за несколько пропусков отлаживаемой программы на машине.

В любом случае основным условием быстрой локализации ошибок является четкое представление о структуре отлаживаемой программы, взаимосвязи блоков и переменных, особенностях реализации. Необходимо, чтобы программист в нужный момент мог быстро и четко, с любой степенью детализации представить себе вычислительный процесс, соответствующий отлаживаемой программе, и, тем самым, но неверным результатом определить характер ошибки и ее место в программе. Существенную помощь при этом могут оказать блок-схемы или обобщенные алгоритмы, а также хорошо прокомментированная и оформленная программа. Успеху локализации ошибок помогает ведение дневника отладки, в котором фиксируется результа

каждого пропуска программы (блока) на машине, характеризуется отход от правильных результатов, намечается следующий шаг отладки. Если программист работает одновременно над несколькими программами или крупными блоками одной и той же программы, то такой дневник является совершенно необходимым. Полученные распечатки размечают, систематизируют и метят таким образом, чтобы привязать их к записям в дневнике.

Помимо того, что дневник помогает проведению текущей отладки, он используется для систематизации опыта локализации ошибок, фиксации своих типичных ошибок, а также для определения трудоемкости проводимых в ходе отладки работ, что помогает правильно запланировать и организовать разработку программ в будущем.

ГЛАВА 3

ИСПРАВЛЕНИЕ ОШИБОК

3.1. Виды исправлений

Исправлять программу приходится много раз: то вследствие обнаружения ошибок, то для вставки отладочной печати, то вследствие изменений требований заказчика, то в ходе модернизации программы. Поэтому освоение программистом практически доступных для него средств исправлений программы имеет большое значение. Ниже вопросы исправления программ будут излагаться с точки зрения устранения ошибок, найденных в программе.

После того, как ошибка в программе найдена, приступают к ее исправлению. В зависимости от того, принадлежит ли найденная ошибка к этапу алгоритмизации или программирования (кодирования), ее исправление требует различного подхода.

3.1.1. Исправление алгоритма. Если обнаруженная ошибка была допущена на этапе алгоритмизации, то перед тем, как пытаться исправлять программу, в которой найдена ошибка, вносят изменения в алгоритм. Трудность исправления алгоритма в первую очередь состоит в том, что ввиду взаимозависимости различных частей программы, исправление ошибки в одном месте может привести к возникновению ошибок в других местах программы. Успешность преодоления такого рода трудностей в значительной мере зависит от строения программы, от методов ее разработки. Чем более независимы части, из которых состоит программа, тем легче исправлять ее алгоритм; рассмотрению этого вопроса посвящен п. 4.2.5.

Исправление алгоритма должно сопровождаться проверкой и, может быть, прокруткой внесенных изменений и затрагиваемых этими изменениями частей программы. Чем на более поздних стадиях производится исправление алгоритма, тем тщательнее должна быть проверка. Заметим, что уже после внесения исправлений в текст программы, для проверки правильности внесенных исправлений про-

грамма подвергается тестированию, причем для значительных исправлений приходится пропускать и тесты ранее уже прошедшие.

3.1.2. Исправление программы. После исправления алгоритма производится перепрограммирование измененного участка алгоритма; такой порядок внесения изменений в программу позволяет сохранить соответствие текстов алгоритма и программы, что облегчает поиск и исправление последующих ошибок. Кроме того, перепрограммирование производится и при обнаружении ошибок этапа программирования, то есть ошибок, возникших при переводе разработанного алгоритма на выбранный язык программирования.

После того, как программа исправлена на бланке или, что чаще, на распечатке, ее требуется исправить на том носителе, где она хранится. В зависимости от вида носителя — перфокарты или магнитные ленты и диски — применяются различные способы исправлений. Перфокарты являются менее удобным видом носителей для хранения текста программы: они мнутся, перетасовываются при вводе, рассыпаются, скорость их ввода гораздо меньше, чем при использовании дисков или магнитных лент.

При хранении программы на перфокартах обычно применяется непосредственное исправление программы путем замены перфокарт в колоде. Если при перфорации перфокарты не надпечатываются, то их приходится надписывать вручную (хотя бы и неполностью) для возможности дальнейших исправлений программы, хранящейся на перфокартах; ручная нумерация перфокарт менее удобна при необходимости многократных исправлений программы. (В некоторых системах применяется автоматизированный способ, при котором перфокарты исправлений просто подкладываются в конец колоды и перед трансляцией они автоматически заменяют или дополняют ранее введенные карты в соответствии с указанными на перфокартах номерами.)

Если программа хранится во внешней памяти, то для ее исправления применяются специальные средства, имеющиеся в используемой операционной системе. Обычно такие средства вносят исправления в исходный текст программы, представленный в символьной форме, после чего, конечно, требуется перетрансляция исправленного текста. Возможность внесения исправлений в оттранслированную уже программу не получила распространения ввиду трудности ее реализации; в ОС ЕС такая возможность используется только для исправления системных программ.

Для внесения изменений в исходный текст программы применяется обычно один из двух видов исправлений: построчное или контекстное.

Построчное исправление служит для исправления целых строчек символьного текста или некоторых полей в них. Программист указы-

вает номера исправляемых строчек и, может быть, номера позиций в строчках.

Контекстное исправление позволяет заменить на правильный текст один и тот же ошибочный фрагмент текста, встречающийся во всей программе или в указанной ее части.

В ОС ЕС ЭВМ основным средством для исправления исходных программ, хранящихся во внешней памяти, является служебная программа UPDATE, выполняющая построчные исправления (см. п.3.2.1). Средств, предназначенных для контекстных исправлений в ОС ЕС нет, но простейшие контекстные исправления, например, в программах, написанных на PL/I, фортране и алголе, могут быть выполнены с помощью препроцессорных средств языка PL/I (см. п.3.2.2). В отличие от служебной программы UPDATE препроцессорные средства могут вносить исправления в программу непосредственно в ходе трансляции без изменения ее текста на внешних носителях; поэтому исправлениям могут подвергаться даже программы, находящиеся на перфокартах.

3.2. Средства исправления программ

3.2.1. Построчные исправления. Построчные исправления излагаются на примере использования служебной программы UPDATE или, вернее, IEBUPDTE, как она именуется в ОС ЕС ЭВМ [14]. Программа позволяет вставлять, удалять и заменять строки некоторого символьного текста, хранящегося во внешней памяти, в частности, текста исходной программы, на каком-либо алгоритмическом языке. Кроме того, поскольку служебная программа для своей работы требует задания номеров исправляемых строчек текста, в ней предусмотрена также возможность перенумерации всех строчек текста и вывод перенумерованного текста на печать. Номера помещаются в нумеруемый текст и хранятся вместе с ним, занимая в строчке указанные позиции; по умолчанию предполагаются позиции 73÷80. Исправляемый текст должен находиться в последовательном или частично-последовательном (библиотечном) наборе данных с длиной логических записей равной 80 байтам.

Рассмотрим использование программы UPDATE для исправления текста на примере выполнения типовых операций исправления: удаления, вставки и замены строк (последняя операция является комбинацией двух предыдущих).

Пример.

```
//PRAVKA JOB ...  
//      EXEC PGM=IEBUPDTE  
//SYSPRINT DD SYSOUT=A  
//SYSUT1 DD DSNAME=ISHODN,DISP=OLD
```

```

//SYSUT2 DD.DSNAME=ISHODN,DISP=OLD
//SYSIN DD DATA
./ CHANGE NAME=FOR901,LIST=ALL
./ DELETE SEQ1=30000,SEQ2=36000
      перфокарта 00042100
      перфокарта 00042200
      перфокарта 00042300
./ NUMBER SEQ1=65000,NEW1=65100,INCR=100,
      INSERT=YES
      8 перфокарт
./ NUMBER SEQ1=127000,NEW1=127100,INCR=100,
      INSERT=YES
      6 перфокарт
./ DELETE SEQ1=128000,SEQ2=131000
./ ENDUP
/*
//

```

В директивах DD со стандартными именами SYSUT1 и SYSUT2 характеризуются входной (исправляемый) и выходной (исправленный) наборы данных: в примере взят один и тот же библиотечный набор данных ISHODN, который ранее был создан и каталогизирован в ОС. После DD с именем SYSIN располагаются операторы программы UPDATE, среди которых находятся и сами перфокарты, содержимое которых вставляется в исправляемый текст или заменяет отдельные строчки текста; последним оператором должен быть оператор ENDUP. Все операторы программы UPDATE начинаются с сочетания знаков . и / .

Оператор CHANGE определяет имя исправляемого раздела — FOR901; параметр LIST=ALL указывает на необходимость выдачи на печать всего текста исправляемого раздела. Оператор DELETE указывает, что требуется удалить из текста строчки с номерами 30000÷36000. Если предположить, что текст был пронумерован с шагом 1000, то будет вычеркнуто 7 строчек (может быть, и меньше, если ранее уже вычеркивались строчки, или больше, если шаг был равен, например, 100). Три перфокарты с пробитыми на них номерами будут вставлены в такое место текста, чтобы последовательность его нумерации не нарушалась, например, после строчки с номером 42000. Если в тексте уже есть строчки с номерами, равными заданным на перфокартах, то произойдет замена старых строчек. Оператор NUMBER с параметром INSERT=YES («ВСТАВИТЬ=ДА») используется для того, чтобы после строчки с номером 65000 вставить 8 перфокарт и перенумеровать их, начиная с 65100 и с шагом 100 (предполагается, что перфокарты не содержат номеров). После выполнения последую-

ших операторов NUMBER и DELETE заменяются строчки 128000-131000 (4 строчки) на содержимое 6 перфокарт, с нумерацией последних от 127100 до 127600.

При выполнении исправлений необходимо следить за тем, чтобы после исправлений последовательность нумерации не нарушалась. Кроме того, и операторы программы UPDATE должны быть расположены в таком порядке, чтобы последовательность номеров обрабатываемых ими строчек текста являлась возрастающей. Средством, которое облегчает выполнение этих условий, является задание шага нумерации большей единицы (см. INCR в приведенном примере).

Помимо изменения разделов (оператор CHANGE) программа позволяет копировать (REPRO) разделы из одного набора в другой, заменять целиком (REPL) содержимое некоторого раздела и дополнить (ADD) набор новым разделом, содержимое которого задано на перфокартах. В последних двух случаях DD с именем SYSUT1 не задается, а перфокарты, составляющие содержание раздела, размещаются среди директив программы UPDATE. Приводимый ниже пример демонстрирует первоначальную запись текста (подпрограммы) в библиотечный набор в качестве его раздела.

```
//CREATE JOB ...
//      EXEC PGM=IEBUPDTE,PARM=NEW
//SYSPRINT DD SYSOUT=A
//SYSUT2  DD DSN=ISHODN,DISP=OLD
//SYSIN   DD DATA
./      ADD NAME=FORS901,LIST=ALL
./      NUMBER NEW1=1000,INCR=1000
           перфокарты исходной программы
./      ENDUP
/*
//
```

При создании раздела необходимо задать PARM=NEW в директиве EXEC, вызывающей программу UPDATE. В примере вводимые перфокарты нумеруются с номера 1000 и с шагом 1000.

По программе UPDATE в одном шаге может быть осуществлено исправление сразу нескольких разделов библиотеки, для чего, например, в предыдущих примерах перед ENDUP можно добавить аналогичные операторы с NAME=FORS902, NAME=FORT008 и т. д.

Имеется возможность исправлять не всю строчку текста, а только отдельное ее поле, в этом случае в исправляемый текст переносится содержимое не всей перфокарты, а лишь указанное ее поле. При использовании программы UPDATE исправляемое поле может располагаться только в конце строчки (перфокарты).

В случае необходимости многократно исправленный текст может быть заново перенумерован в любом выбранным шагом.

З а м е ч а н и е. После многократных исправлений разделов в библиотечном наборе он должен быть «почищен», т. е. из него необходимо удалить дубликаты исправленных разделов, оставляемых в наборе программой UPDATE и недоступных пользователю (если ими не были приняты специальные меры, связанные с применением оператора ALIAS); такое сжатие библиотеки, предупреждающее ее переполнение, осуществляется с помощью служебной программы COPY [14].

3.2.2. Контекстные исправления. Ниже на примерах будет показано, как можно использовать препроцессорные средства для осуществления простейших контекстных исправлений в программах, написанных на алгоритмических языках. Препроцессорные средства удобно использовать для замены идентификаторов во всей программе или в некоторой ее части на другие идентификаторы, выражения, операторы и, вообще, на любой фрагмент текста. В отличие от информации для служебной программы UPDATE исправляющая информация для препроцессора обычно должна задаваться вместе с исправляемым текстом программы, т. е. препроцессорные операторы необходимо размещать среди обычных операторов языка, что, конечно, не всегда является удобным. Подробнее препроцессорные средства описаны в Приложении.

П р и м е р. Предположим, что в некоторой программе, кроме отдельных ее участков, требуется заменить идентификатор А на выражение В—5. Пусть исправления относятся к следующему тексту, хранимому на перфокартах:

```

.. . . . .
X=A;

.. . . . .
Y=A+B*C; A=Z;

.. . . . .
Z=FUN (A*4);

.. . . . .

```

Замену необходимо производить начиная с оператора, вычисляющего X, отключаться перед присваиванием Y, и снова включаться перед вычислением Z.

Поставленная задача решается с использованием препроцессорных операторов следующим образом:

```

.. . . . .
% DECLARE A CHARACTER ;

.. . . . .
% A='(B-5)';
  X=A;
  . . . .

```

```
% DEACTIVATE A;
    Y=A+B*C; A=Z;
    . . . . .
% ACTIVATE A;
    Z=FUN (A*4);
    . . . . .
```

Оператор % DECLARE описывает в качестве препроцессорной переменной тот идентификатор, который необходимо изменить. В препроцессорном операторе присваивания задается требуемое исправление идентификатора (переменной) А. Операторы дезактивации и активации выключают и включают для последующего текста замену идентификатора А.

Описание препроцессорной переменной А может быть расположено и в самом начале программы, например, перед заголовком главной процедуры.

Препроцессорный оператор присваивания А должен быть расположен после оператора описания А, но перед оператором PL/1, содержащим исправляемое А. Круглые скобки добавлены для того, чтобы выражения типа А*4 получили правильный вид: (В—5)*4, а не В—5*4. Если бы нужна была замена во всей программе, то операторы дезактивации и активации являлись бы излишними.

С помощью препроцессорных средств заменяются любые идентификаторы, в частности, можно осуществить, например, и следующую замену

$\text{SIN (H)} \rightarrow \text{EXP (H)} * \text{COS (H)}$

выполнив препроцессорный оператор присваивания

%SIN= 'EXP (H)*COS';

Контекстная замена выражений или операторов, содержащихся в исходном тексте, является невозможной для препроцессорных средств.

Исправление программ на алголе или фортране производится аналогичным образом с использованием тех же препроцессорных операторов; ограничения см. в П.7.

Если исправляемый текст находится на перфокартах, то для его замены перфокарты с препроцессорными операторами подкладываются в нужные места программы. Если же исправляемый текст находится во внешней памяти, то препроцессорные операторы, осуществляющие замену, могут быть внесены в текст программы по служебной программе UPDATE.

Если исправляемый текст хранится во внешней памяти в разделе некоторой библиотеки и включается в текст программы с помощью

препроцессорного оператора `% INCLUDE`, то при условии, что контекстные исправления должны быть проведены во всем разделе, они могут быть реализованы двумя препроцессорными операторами (описания и присваивания), аналогичными тем, которые были приведены выше. Например, пусть приведенный ранее текст хранится в разделе FON библиотеки, характеризующейся в DD-директиве с именем PRELIB. Тогда его исправление реализуется следующей последовательностью операторов, располагаемых в нужном месте программы:

```
% DECLARE A CHAR ;  
% A='(B—5)';  
% INCLUDE PRELIB (FON);
```

Первые два оператора могут быть расположены и в начале программы, содержащей оператор `% INCLUDE`; сама эта программа может находиться во внешней памяти и, в свою очередь, включаться в текст основной программы по `% INCLUDE`.

Как уже упоминалось ранее, исправление какой-либо программы препроцессорными средствами происходит непосредственно перед ее трансляцией, и при этом внесение исправлений в текст программы, находящейся на каком-либо носителе, не производится (см. также П.1). Для того чтобы действительно изменить текст программы, расположенной на внешнем носителе (диске или магнитной ленте) необходимо использовать специальные параметры транслятора, например, такой как MACDCK, задающий вывод исправленного текста на внешний носитель; характеристики выводимого набора данных указываются в директиве DD с именем SYSPUNCH (см. П.7).

3.3. Классификация видов исправлений

Ниже дается классификация видов исправлений, упомянутых выше.

Исправления:

1. Алгоритма.
2. Исходной программы.
 - 2.1. На бланке (непосредственное).
 - 2.2. На перфокартах.
 - 2.2.1. Ручная замена перфокарт.
 - 2.2.2. Автоматизированная замена.
 - 2.2.2.1. Построчная
 - 2.2.2.2. Контекстная (по препроцессору).
 - 2.3. Во внешней памяти.
 - 2.3.1. Построчные (по UPDATE).
 - 2.3.2. Контекстные (по препроцессору).
3. Объектной программы.

ГЛАВА 4

ПРЕДУПРЕЖДЕНИЕ ОШИБОК

Предыдущие главы касались вопросов обнаружения и исправления ошибок и в них рассматривались те средства и методы, которые способствуют быстрейшему выполнению таких работ на этапе отладки. Но к вопросу ликвидации (или, точнее, уменьшения) ошибок в программе можно подойти и с другой стороны, рассматривая методы программирования, предупреждающие появление ошибок в программе или облегчающие их поиск и исправление на последующих этапах.

Сначала будут рассмотрены широко распространенные приемы и способы, позволяющие уменьшить количество ошибок в программе, а затем — основы современных методов разработки: модульного и структурного программирования; в следующей главе на примерах разбирается метод нисходящего проектирования.

4.1. Традиционные приемы

4.1.1. Этапность разработки. Выше (см. Введение и гл. 1) уже отмечалось, что разбиение процесса разработки программы на этапы, письменная фиксация итогов выполнения всех этапов и проверка этих итогов на каждом этапе содействуют обнаружению многих ошибок в ходе разработки программы и препятствуют их проникновению в готовую программу. Напомним о таких этапах, как составление технического задания и проекта программы, разработка алгоритма и формализованный переход от алгоритма к программе, проверка алгоритма и программы за столом и т. п. На рис. 5 схематически показаны два разных подхода к разработке программы. В первом случае разработка состоит из двух этапов: составление программы (программирование) и ее отладка. Во втором случае перед составлением программы производится ее поэтапное проектирование, и часть отладочных работ проводится в ходе такого проектирования.

Этапность разработки программы позволяет программисту сосредоточить все свое внимание на узкоспециализированной работе при выполнении каждого отдельного этапа. Узкая специализация позволяет снизить напряженность в работе и облегчить тем самым ее выполнение, что, в свою очередь, должно привести к уменьшению количества ошибок, вносимых во время такой работы.

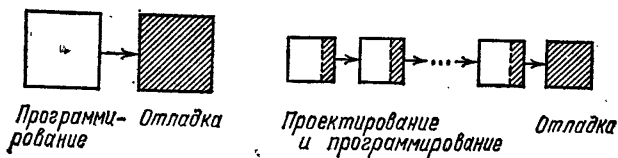


Рис. 5. Этапность разработки программы.

4.1.2. Уровень языка. Программирование на языках высокого уровня (из широкоизвестных — PL/1, алгол-60, в известной мере, фортран) позволяет формулировать решение задачи в обозначениях, приближенных к обычному математическому языку, что способствует уменьшению количества ошибок в программе, так как позволяет программисту отвлечься от множества деталей, которые необходимо учитывать в языках низкого уровня (автокоды, ассемблеры и особенно машинные языки). Кроме того, более крупные операторы языков высокого уровня более наглядно отражают логическую структуру программы и, тем самым, предупреждают возникновение логических ошибок. Но, пожалуй, еще более важно, что отмеченные свойства программ на языках высокого уровня значительно облегчают их проверку, а также поиск ошибок и внесение исправлений в ходе отладки.

К языкам самого высокого уровня можно отнести и языки макросредств, например, таких как препроцессорные средства PL/1 (см. Приложение) и макросредства языка ассемблера ЕС ЭВМ. Макросредства позволяют программисту в ходе написания программы оперировать не только с отдельными операторами, но и с произвольными совокупностями операторов (или частями операторов) и составлять из этих совокупностей программу, модифицируя их при этом необходимым образом.

Из вышесказанного следует, что нужно стремиться писать большую часть программы на языках высокого уровня, используя и имеющиеся макросредства. В случае необходимости для достижения большей эффективности программирование некоторых блоков программы приходится производить на языке ассемблера (после трансляции такие блоки объединяются с основной частью программы с помощью редактора связей). Объем таких блоков обычно составляет неболь-

шую долю от объема всей программы: по статистическим данным более 50% общего времени работы программы приходится на выполнение лишь 5% команд типичной программы, и именно эти 5% нужно найти и запрограммировать на ассемблере.

4.1.3. Наглядность текста программы. Подавляющая часть времени уходит у разработчика не на то, чтобы писать программу, а на ее чтение: при проверке программы, поиске ошибок, внесении исправлений в ходе отладки или дальнейшей эксплуатации. Поэтому необходимо позаботиться о том, чтобы текст программы обладал свойством наглядности и читабельности. Даже при беглом просмотре текста должна выявляться структура программы, соподчиненность составляющих ее компонент — операторов, блоков. В частности, следует придерживаться следующих рекомендаций:

- располагать описания в начале программы и первый выполняемый оператор помечать специфической меткой;

- END (end) располагать в тех же колонках, в которых находятся соответствующие BEGIN (begin), DO, PROCEDURE (procedure);

- операторы, являющиеся частями других более крупных операторов, располагать со смещением на несколько позиций вправо — «лесенкой»;

- в каждой строчке текста размещать, как правило, по одному оператору, за исключением случаев, когда операторы небольшие и логически тесно связаны между собой (такое размещение облегчает и исправление программы путем замены перфокарт или построчными средствами);

- метки, как правило, должны находиться в самых левых из занятых позиций строчки (или даже в самых первых позициях строчки) они не должны «загораживаться» слева другими операторами, как, например, в следующем примере:

```
IF T<1000 THEN GAS=1; BACK: TEPLO=TEPLO+23;
```

- для облегчения чтения текста следует использовать и пробелы, выделяя с их помощью метки, операторы, отдельные их части, а также операции младших рангов в сложных выражениях; пустыми строчками полезно отделять одни блоки или группы операторов, логически связанные между собой, от других;

- идентификаторы должны быть mnemonicны, они должны напоминать программисту о смысле обозначаемых величин, например;

```
VELOCITY или VELO  
PRESS или DAVLENIE
```

- правила умолчания следует применять весьма осторожно, например, только в случае, когда они разрешаются стандартами программирования для данной разработки (см. ниже 4.1.5).

Одним из основных средств, проясняющим смысл различных фрагментов программы, являются комментарии. Следует, пользуясь комментариями, допустимыми в конкретном языке программирования, отмечать основные этапы программы, указывать содержательный смысл величин при их описании, пояснять наиболее трудные для понимания участки программы, чтобы во время отладки быстро ориентироваться в них. Комментарии должны также помогать установлению соответствия между операторами программы и обобщенного алгоритма (блок-схемы); при этом для краткого комментирования можно использовать и метки языка, располагая их, в частности, на отдельной строчке. В ходе разработки при исправлении программы и ее алгоритма нужно следить за тем, чтобы комментарии всегда соответствовали ее тексту.

Если программа предназначается для длительного использования, особенно другими программистами или пользователями, а также для того, чтобы облегчить последующую модернизацию программы, ее нужно снабжать гораздо более подробными комментариями. Рекомендуются, помимо одного комментария на 2-3 строчки текста, описывать также назначение блока (процедуры) и его входные и выходные данные. Комментарии должны быть легко понимаемы другими программистами.

Логическим завершением хорошо прокомментированной программы является *самодокументированная* программа, т. е. программа, объединенная со своим описанием, имеющим вид комментариев. Самодокументированная программа снабжается всей информацией, необходимой для ее последующей отладки и практического использования в дальнейшем, в частности, для ее изучения и дальнейшего развития. Текст самодокументированной программы хранят во внешней памяти, что позволяет легко вносить исправления в изготавливаемую программу и ее описание в ходе разработки программы; пользуясь служебными программами, можно в нужные моменты выдавать на печать или всю программу, или отдельные страницы после их исправления. Самодокументированные программы особенно полезны при крупных и длительных разработках.

4.1.4. Личные приемы. Опытный программист в практической работе использует обычно лишь небольшое количество приемов программирования, причем таких, которые привычны ему настолько, что проникновение ошибок реализации в готовую программу при этом практически исключается. При необходимости применения новых, мало знакомых приемов программирования следует уделять больше внимания (как на этапе проверки программы, так и при ее тестировании) тем местам программы, где применялись такие новые приемы; во время локализации ошибок такие участки следует всегда держать под подозрением. Следует избегать очень сложных и тонких способов

программирования, требующих особого внимания при их применении — нужно помнить, что, «где тонко, там и рвется»! И дело даже не в том, что увеличивается вероятность внесения ошибок, а в том, что при этом обычно требуется много усилий и машинного времени для того, чтобы доказать их отсутствие.

Каждый программист должен иметь осознанный набор личных приемов программирования, совершенствовать их и пополнять новыми, подвергая такие приемы предварительному опробованию. При программировании не следует полагаться только на свою память, а для предупреждения ошибок и ускорения программирования использовать разного рода таблицы, в которых собраны шаблоны применяемых приемов программирования и основная информация, необходимая при работе.

4.1.5. Стандарты программирования. Большинство составляемых программ является частями какой-то программной системы, а программисты являются членами некоторого единого коллектива программистов во главе с руководителем. Поэтому для упрощения стыковки и взаимодействия отдельных блоков программистами, а также для облегчения проверки и контроля этих блоков со стороны старших программистов вводятся стандарты программирования. В стандартах могут быть установлены типовые приемы программирования и организации данных, допускаемые размеры блоков (модулей) и способы их взаимосвязи, правила оформления блок-схем и описаний программ, способы выдачи диагностики и т. п.

Стандарты могут меняться от группы к группе, поскольку они зависят и от характера задач и от вкусов руководителей. Хорошие стандарты, несомненно, ускоряют и облегчают разработку программ и уменьшают количество ошибок в них. Они освобождают программиста от ненужных размышлений и экспериментов над второстепенными вопросами и позволяют ему сосредоточить свое внимание на наиболее принципиальных или трудных вопросах.

4.1.6. Защита от ошибок. Для быстрой локализации ошибок в отлаживаемой программе в текст программы вносят операторы, в задачу которых входит обнаружение ошибочных результатов, возникающих в ходе выполнения программы. Такие результаты могут явиться следствием как ошибок в тексте некоторого блока программы, так и ошибочных данных, передаваемых этому блоку из других блоков. Для того, чтобы предупредить ошибки в данных, следует контролировать данные, являющиеся входными для блока, например, в начале процедуры проверять значения полученных аргументов, а в ходе выполнения блока — данные, считываемые с магнитной ленты или диска и т. п. При выполнении программы имеет смысл также проверять получаемые значения некоторых величин на их соответствие установленным диапазонам, например, это относится к индексам

сам, к переменной переключателя (вычисляемого GO TO в фортране) и т. п. Конечно, речь идет о проверке значений не всех величин, а только узловых и, кроме того, удобных для проверки.

Операторы, защищающие от ошибок, оказываются полезными не только при отладке, но и на этапе счета по отлаженной программе для обнаружения ошибок в исходных данных программы или в ответах оператора ЭВМ на запросы программы. Кроме того, они служат для выявления ошибочных результатов, которые могут быть получены в результате сбоя ЭВМ или из-за отказа какого-либо внешнего устройства. Желательно обнаружить такие ошибки как можно раньше, пока ошибочные результаты не переданы другим частям программы. После обнаружения ошибки выдается диагностика, вычисления прекращаются или делается попытка повторить вычисления.

Одни операторы, реализующие защиту от ошибок, могут постоянно находиться в программе, другие удаляются из нее после окончания отладки. В последнем случае вставку и устранение операторов защиты удобно производить с помощью макросредств, например, используя препроцессорные операторы PL/1 (см. ниже П.4).

Планирование защиты от ошибок проводится на этапах проектирования и алгоритмизации, а реализация осуществляется на этапе программирования (кодирования). Для реализации защиты от разного рода ошибок (в частности, для контроля изменения индексов) в PL/1 могут быть использованы исключительные ситуации, а в фортране и алголе ЕС отладочные режимы (см. 2.1.5).

Другим частным случаем защиты от ошибок, предусмотренным в PL/1, является контрольный список меток, задаваемый при описании меточной переменной: в случае присвоения такой переменной меточного значения, отличного от указанных в списке, во время выполнения программы фиксируется ошибка. При отсутствии такого списка меток оператор перехода (GO TO), содержащий меточную переменную, направил бы вычислительный процесс по неверному пути, и обнаружить такую ошибку другими средствами было бы весьма затруднительно.

С помощью операндов TIME или OUTLIM в директивах JOB, EXEC или DD языка управления заданиями ОС ЕС ЭВМ следует ограничивать время выполнения задания или шага и количество печатаемой информации, что окажется особенно полезным при возникновении заикливания в выполняемой программе.

Конечно, защита от ошибок замедляет выполнение программы и, оставляя операторы защиты на этап счета, нужно быть уверенным, что это замедление не окажется слишком большим.

4.2. Модульность программы

4.2.1. Модульность. Модульной называют программу, составленную из таких частей — *модулей*, что их можно независимо друг от друга программировать, транслировать, отлаживать (проверять, исправлять). Предполагается, что модули имеют небольшие размеры, четко определенные функции и, кроме того, их связи между собой максимально упрощены, в частности, предполагается, что модули имеют лишь одну точку входа (в начале модуля). Разбиение программы на модули, осуществляемое на стадиях проектирования и алгоритмизации, хотя и является весьма непростым делом, позволяет существенно облегчить в дальнейшем работу над программой на различных этапах.

После того, как в алгоритме выявлены мало зависимые друг от друга части, составление программы упрощается, так как при программировании каждой из этих частей почти не приходится заботиться об их взаимодействии с другими частями, что в свою очередь способствует уменьшению количества вносимых ошибок. Кроме того, малая зависимость модулей позволяет при необходимости существенно распараллелить составление программы, поручив программирование разным программистам, причем всегда можно найти подходящую работу и для начинающих и для опытных программистов.

На этапе отладки независимость модулей позволяет отлаживать их в любом порядке, в частности, и одновременно. Считается, что усилия, затрачиваемые на отладку модуля, обычно пропорциональны квадрату его длины [1, VIII], и поэтому при тестировании небольшие размеры модулей дают возможность поставить задачу о проверке в *с е х* ветвей таких модулей, что ведет к увеличению достоверности тестирования. Решение такой задачи является обычно недостижимым по отношению ко всей программе или крупным ее блокам, когда приходится ограничиваться лишь проверкой работы всех линейных участков блока и условий (см. 1.2.2). Разумеется, и наиболее трудная часть отладки — локализация ошибок, проводимая для модулей, при этом значительно упрощается и ускоряется.

В силу минимальности логических связей между модулями облегчается, конечно, и внесение исправлений в алгоритм программы, поскольку меньше приходится заботиться о том, чтобы при изменении одной части программы не испортить работу другой ее части (подробнее об этом будет сказано в 4.2.5).

4.2.2. Реализация модульности. Модули для программ, реализуемых на алгоритмических языках PL/1, фортран, алгол-60, наиболее естественно представлять в виде внешних процедур (ранее оттранслированных процедур — для алгола ЕС, подпрограмм — для фортрана). Помимо того, что их можно независимо составлять, про-

верить и отлаживать, их можно также независимо транслировать и исправлять. Внутренние процедуры не обеспечивают такой независимости реализуемых модулей: их нельзя независимо транслировать и исправлять.

Информационную связь между модулями-процедурами для максимальной их независимости имеет смысл осуществлять только с помощью аргументов при обращении к процедуре (по оператору CALL в PL/1, фортране и ассемблере или по оператору процедуры в алголе-60, а также по указателям функций). На практике, в силу значительных трудностей при выделении подлинно независимых модулей, приходится прибегать и к другим, дополнительным формам связи модулей таким, как:

- передача информации в использовании внешних переменных (EXTERNAL, COMMON) — для PL/1 и фортрана;
- передача информации с помощью глобальных переменных (для внутренних процедур) — для PL/1 и алгола;
- вход внутрь процедуры по дополнительному входу, указанному оператором ENTRY — для PL/1 и фортрана.

Таким образом, приходится говорить о той или иной степени модульности программы, о большей или меньшей независимости составляющих ее частей. Например, по степени независимости модулей-процедур можно различить следующие 4 вида модулей:

- а) внешняя процедура без дополнительных связей;
- б) внешняя процедура с дополнительными связями;
- в) внутренняя процедура без дополнительных связей;
- г) внутренняя процедура с дополнительными связями.

Что касается рекомендуемых размеров модуля, то различные авторы приводят разные данные — от нескольких десятков, до нескольких сотен операторов в модуле. В некоторых организациях считают, что есть смысл добиваться того, чтобы модуль занимал не более одной страницы при печати, то есть чтобы, например, при работе в ОС ЕС ЭВМ все операторы модуля размещались на 60 строчках АЦПУ.

Чем более мелкими требуется получать модули, тем больше трудностей возникает при проектировании и алгоритмизации программы, но тем легче будет каждый из модулей проверять и тестировать в дальнейшем. Не следует, однако, забывать и о том, что слишком большое количество мелких модулей может значительно увеличить трудоемкость предстоящей комплексной (стыковочной) отладки.

Если разработанный модуль не уместается в установленные размеры, то из него следует выделить одну или несколько достаточно независимых частей, которые желательно оформить в виде внешних процедур, вынеся их из модуля, а на их места вставить операторы вызова выделенных модулей-процедур. Если модули получаются мелкими, они могут иметь вид внутренних процедур или даже блоков,

объединяемых во внешнюю процедуру; внутренние процедуры обычно располагаются в начале внешней.

З а м е ч а н и е. Серьезной помощью в разработке программ могут стать библиотеки стандартных или типовых модулей, заранее составленных автором или другими программистами. Применение при разработке ранее многократно опробованных модулей, трудность использования которых сводится только к заданию правильных аргументов, значительно ускоряет составление программы и облегчает ее отладку. Отметим, что в библиотеку могут быть организованы и препроцессорные («макро») модули, включаемые в разрабатываемую программу по оператору включения (см. ниже 4.2.4).

4.2.3. Строеение программ. Не претендуя на полноту классификации, строение программ можно охарактеризовать одной из следующих схем, представленных на рис. 6 (стрелки указывают на связи модулей: вызов, передача данных, возврат).



Рис. 6. Строеение программ.

1. Монолитное. Программа написана цельным куском, без выделения каких-либо отдельных независимых частей.

2. Монолитно-модульное. Имеется достаточно большая монолитная главная часть программы, в которой производятся основные вычисления, и из которой происходят последовательные обращения к модулям.

3. Последовательно-модульное. Центральная часть программы состоит из последовательно выполняемых модулей, которые в свою очередь обращаются к другим модулям.

4. Иерархическое. Программа состоит из модулей, связи между которыми подчиняются строгой иерархии: каждый модуль может обращаться только к модулям, которые ему непосредственно подчинены. Возврат всегда должен происходить в вызывающий модуль, даже в том случае, если в вызываемом модуле обнаруживается ошибка, препятствующая дальнейшим вычислениям (не все языки, правда, имеют средства для выполнения этого требования).

5. Иерархически-хаотическое. Иерархическая (или последовательная) подчиненность модулей нарушена дополнительными связями (штриховые стрелки).

6. Модульно-хаотическое. Программа состоит из модулей, но связи их между собой не отвечают принципу иерархии (или последовательности).

Последовательно-модульное и иерархическое (для более сложных программ) строения, как наиболее простые по логическим связям, являются теми образцами, к которым необходимо стремиться при разработке программы. Допустимыми вариантами являются иерархически-хаотическое и, может быть, монолитно-модульное.

Ниже специфика разработки модульных программ будет рассмотрена более подробно (см. также гл. 5).

4.2.4. Трудности. Представление разрабатываемой программы в виде суммы модулей, удовлетворяющих всем перечисленным ранее требованиям, является достаточно сложной работой. Уже на этапах проектирования и алгоритмизации программисту необходимо разбить алгоритм на мало зависящие друг от друга части, выявить связывающие величины и установить способы передачи их значений между модулями; желательно, чтобы количество таких связывающих величин было минимальным. Для успеха модульного программирования необходимо, чтобы программист приобрел определенные навыки в такого рода работе, когда уже на начальных стадиях разработки программы приходится решать наиболее сложные и важные вопросы организации будущей программы. Иногда добиться необходимого уровня модульности программы бывает очень сложно, так как реальные связи в реализуемом алгоритме препятствуют этому, и приходится жертвовать некоторой долей независимости и, тем самым, некоторыми удобствами работы с программой и ее модулями в будущем.

Кроме того, приходится учитывать, что модульность может снизить эффективность работы программы. Например, в некоторых трансляторах обращение к процедуре с передачей аргументов осуществляется настолько неэффективно, что в качестве модулей приходится рассматривать простые, а не процедурные блоки (в PL/1 и алголе), а информационные связи осуществлять через глобальные переменные. В ряде случаев для небольших модулей может оказаться выгодным, вместо обращения к модулю, просто включить его текст в программу (в виде «открытой» подпрограммы). В ОС ЕС такое включение удобно осуществлять, используя препроцессорный оператор % INCLUDE (см. П.6).

П р и м е р. Предположим, что в некоторой задаче необходимо часто менять местами значения различных величин, выполняя последовательность операторов следующего вида:

$$P := X; X := Y; Y := P$$

где X и Y — изменяемые величины, а P — рабочая переменная (здесь даны операторы, удовлетворяющие синтаксису алгола ЕС).

Вместо того, чтобы оформлять эту группу операторов в процедуру и обращаться к ней каждый раз, можно использовать препроцессорные средства. Для этого в библиотеку исходных модулей записы-

вается раздел, содержащий указанные выше или следующие операторы:

```
'BEGIN' P :=X; X :=Y; Y :=P 'END';
```

Имена X и Y в модулях, где применяются перестановки, должны быть описаны как препроцессорные переменные:

```
% DECLARE (X, Y) CHARACTER;
```

В тех местах, где необходимо осуществлять перестановки, нужно, используя препроцессорные операторы присваивания, изменить имена этих переменных (ранее взятые произвольно) на имена конкретных переменных (ср. с п.3.2.2).

В тексте процедуры, приведенной ниже, заготовка, переставляющая значения, используется для транспонирования матрицы и для изменения на обратный порядок следования элементов в векторе. Ниже в примере предполагается, что:

TRANS — имя раздела, содержащего заготовку для перестановки значений (см. выше составной оператор);

BIBTR — имя директивы DD, содержащей характеристики библиотечного набора данных, используемого для хранения раздела TRANS.

```
'PROCEDURE' MODULE (A, B, N);  
  'ARRAY' A, B; 'INTEGER' N; 'VALUE' N;  
% DECLARE (X, Y) CHARACTER;  
  
  .....  
  'BEGIN' 'REAL' P; 'INTEGER' I, J, K;  
    'COMMENT' /*ТРАНСПОНИРОВАНИЕ МАТРИЦЫ A*/;  
    'FOR' I:=1 'STEP' 1 'UNTIL' N 'DO'  
      'FOR' J:=1 'STEP' 1 'UNTIL' I-1 'DO'  
% X='A (/I, J)'; % Y='A(/J, I)';  
% INCLUDE BIBTR (TRANS);  
% /*'BEGIN' P:=A[I, J]; A[I, J]:=A[J, I];  
   A[J, I]:=P 'END' */;  
  
  .....  
  'COMMENT' /* ПЕРЕВЕРТЫВАНИЕ ВЕКТОРА B */;  
  'FOR' K:=1 'STEP' 1 'UNTIL' N '/' 2 'DO'  
% X='B(/K)'; % Y='B(N-K+1)';  
% INCLUDE BIBTR (TRANS) /* B[K] <=> B[N-K+1] */;  
  
  .....  
  'END' /* БЛОКА И МОДУЛЯ */
```

Конечно, наглядность алгоритма при использовании препроцессорного оператора % INCLUDE значительно снижается, и поэтому следует давать комментарии; в примере препроцессорные ком.

ментарии содержат операторы, соответствующие генерируемым с помощью оператора % INCLUDE и предшествующих препроцессорных операторов присваивания. Квадратные скобки (которые обычно входят в набор символов ЕС ЭВМ) заменяют в комментариях сочетания (/ и /), используемые в алголе ЕС в качестве индексных скобок. Напомним, кроме того, что сочетание '/' служит для представления операции целочисленного деления (\div), а /* и */ в алгольных комментариях необходимы при использовании препроцессорных средств (см. П.7). Упоминаемые ранее комментарии включены в тело препроцессорного оператора (пустого или % INCLUDE), и поэтому в транслируемом тексте они будут отсутствовать.

Применение препроцессорных процедур позволяет предложить другой более удобный и более реальный вариант решения поставленной задачи. Ниже приводится пример для случая, когда требуется получить текст модуля на языке PL/1.

```
MODULE: PROCEDURE (A, B, N);
    DECLARE A(*,*), B(*), N, P;
    % DECLARE TRANS ENTRY (CHAR, CHAR)
                                     RETURNS (CHAR);
    % TRANS: PROCEDURE (X, Y) RETURNS (CHAR);
        DECLARE (X, Y) CHAR;
        RETURN('DO; P=' || X || ';'
              || X || '=' || Y || ';'
              || Y || '=' || P; END;');
    % END TRANS:

    . . . . .
    /* ТРАНСПОНИРОВАНИЕ МАТРИЦЫ А */
    DO I=1 TO N;
        DO J=1 TO I-1;
            TRANS (A(I, J), A(J, I))
            % /* A[I, J] <=> A[J, I] */;
        END;
    END;

    . . . . .
    /* ПЕРЕВЕРТЫВАНИЕ ВЕКТОРА В */
    DO K=1 TO N/2;
        TRANS(B(K), B(N-K+1))
        % /* B[K] <=> B[N-K+1] */;
    END;

    . . . . .
END MODULE;
```

Результатом работы препроцессорной процедуры-функции TRANS является символьная строка, получаемая путем сцепления (конкатенации — ||) символьных констант и значений препроцессор-

ных переменных X и Y. Обращение к препроцессорной процедуре-функции TRANS имеет вид указателя функции и не содержит ни ключевого слова CALL, ни точки с запятой. Аргументы в обращении здесь считаются значениями символьных строк-констант и поэтому в апострофы не заключаются (см. П.5). Комментарии к обращениям содержатся в пустом препроцессорном операторе, чтобы они не попали в текст программы, направляемый на трансляцию.

Заметим, что препроцессорные процедуры могут быть только внутренними: они известны лишь внутри содержащего их модуля, и в этом состоит их ограниченность по сравнению с оператором % INCLUDE. Для того, чтобы можно было использовать препроцессорные процедуры, расположенные вне тела препроцессорной процедуры, приходится прибегать к предварительной вставке их в тело программы с помощью оператора % INCLUDE.

Использование препроцессорной процедуры или оператора включения вместо обычной процедуры позволяет значительно сокращать время работы циклов (особенно вложенных), поскольку исключает многократное выполнение оператора CALL, каждая реализация которого занимает несколько десятков машинных команд. В наших примерах, поскольку в заготовке и процедуре TRANS содержится всего 4 формируемых переменных (с именами X и Y), их использование в данном конкретном случае мало облегчает работу программиста (он мог бы и просто вписать текст соответствующих операторов в тело операторов цикла), но если таких изменяемых имен в формируемых операторах содержится несколько десятков, то применение препроцессорных средств значительно упрощает программирование в подобных случаях.

4.2.5. Локальность исправлений. Рассмотрим теперь более подробно особенности работ, связанных с исправлениями, производимыми в модульной программе. Будем говорить, что программа обладает свойством *локальности исправлений*, если при внесении какого-либо изменения в алгоритм некоторого модуля, исправления необходимо производить только в этом модуле программы.

Исправления могут выполняться как на стадии отладки программы, так и на дальнейших этапах: при эксплуатации или модернизации программы. Причем, если при отладке программы исправления касаются, в основном, деталей алгоритма, то при эксплуатации программы исправления могут касаться и более общих сторон алгоритма, чаще всего таких как форматы внешних данных и значений величин, являющихся параметрами программы (принятых ранее, во время составления задания на программирование, за константы). Ниже мы рассмотрим различные виды изменений программы и такие способы составления программ, которые позволяют локализовать предстоящие исправления внутри соответствующего модуля.

Крупные исправления. Из требуемой четкой функциональности всех модулей следует, что серьезное изменение какой-либо из функций программы при модернизации должно свестись к изменению или, может быть, к замене целиком одного из модулей программы (или нескольких модулей при особенно крупном изменении). Значительные дополнения к программе (или к какому-нибудь модулю) имеет смысл оформлять в виде отдельных модулей.

Изменение форматов данных. Список и форматы вводимых и выводимых данных, как правило, подвергаются изменениям в ходе отладки и пробной эксплуатации программы. Конечно, это свидетельствует о недостаточной продуманности задания на программирование или общего проекта, но фактом является то, что обычно после получения первых результатов появляются и новые пожелания, и дополнительные требования пользователей, касающиеся изменения форм общения с программой. Для того, чтобы сделать внесение требуемых изменений менее болезненным, стараются выделять специализированные модули ввода и вывода, и в них сосредотачивать весь аппарат общения с внешней средой.

Иногда к программированию ввода-вывода подходят с требованием независимости модулей от вида внешних носителей, что, например, дает возможность, не меняя текста модуля работать при отладке программ с дисками или перфокартами, а во время эксплуатации — с магнитными лентами. В ОС ЕС ЭВМ эта задача для последовательных наборов данных решается вынесением форматов внешних данных из программы в директивы DD языка управления заданиями, где они могут подвергаться изменениям независимо от текста программных модулей.

Изменение констант. Для того, чтобы программа меньше подвергалась исправлениям при изменении ее констант, следует заменять переменными те из констант, которые могут измениться в ходе эксплуатации и модернизации. Перед использованием таких переменных в программе им необходимо присвоить значение, равное заменяемой константе. Присвоение может осуществляться, как обычным оператором присваивания, так и другими, имеющимися в языке средствами, например, в PL/1 это можно осуществить с помощью атрибута INITIAL; кроме того, могут быть использованы и препроцессорные средства ОС ЕС. При замене констант на переменные, следует обратить внимание на мнемоничность заменяющих идентификаторов, иначе наглядность программы может уменьшиться.

Частным, но очень важным случаем параметров-констант программы являются размеры таблиц, задаваемые граничными параметрами при описании массивов в программе. Если не ввести переменные для обозначения границ индексов, то в дальнейшем при изменении размеров таблиц, может быть, потребуется вносить исправления в текст

везде, где встречается, например, максимальное значение индекса: в заголовках цикла, в индексе переменных и т. п.

В том случае, когда изменяемые параметры программы используются в разных модулях, возникает задача такого построения программы, чтобы начальные значения параметров можно было изменить лишь в одном модуле. Этого можно добиться использованием описателей EXTERNAL (в PL/1) или COMMON (в фортране) для одних и тех же величин, встречающихся в различных модулях. Исходное значение таким величинам может быть присвоено в одном из обычных модулей, но удобнее выделить специализированный модуль, где и собрать все присваивания значений параметрам программы.

Пр и м е р. Пусть имеется программа, состоящая из модулей A и FUN:

```
A: PROCEDURE OPTIONS (MAIN);  
    DECLARE (U, Q) FLOAT (10);  
    . . . . .  
    U=3.456*P+FUN(Q)/1.289;  
    . . . . .  
END A;  
  
FUN: PROCEDURE (X) RETURNS (FLOAT (10));  
    DECLARE (R, X) FLOAT (10);  
    . . . . .  
    RETURN (2.34*X*R+1.728);  
END FUN;
```

В ходе разработки программы выяснилось, что коэффициенты 3.456 и 2.34 в дальнейшем могут изменяться, и поэтому требуется так перестроить программу, чтобы тексты модулей A и FUN не менялись, при изменении указанных коэффициентов. Ниже приводятся результаты одной из возможных перестроек.

```
PARAM: PROCEDURE OPTIONS (MAIN);  
    DECLARE (TEPLO INITIAL (3.456),  
            PRESS INIT (2.34)) EXTERNAL;  
    CALL A;  
END PARAM;  
  
A: PROCEDURE;  
    DECLARE TEPL0 EXTERNAL;  
    DECLARE (U, Q) FLOAT (10);  
    . . . . .  
    U=TEPLO*P+FUN(Q)/1.289;  
    . . . . .  
END A;
```

```

FUN: PROCEDURE (X) RETURNS (FLOAT (10));
    DECLARE (PRESS, TEPLO) EXTERNAL;
    DECLARE (R, X) FLOAT (10);
    RETURN (PRESS*X*R+TEPLO/2);
END FUN;

```

Задачей головного модуля PARAM является описание параметров программы и присвоение им установленных значений, а также вызов для выполнения модуля А. Введенные параметры TEPLO и PRESS описываются во всех модулях как внешние переменные. В больших комплексах модулей общие описания внешних переменных удобно оформлять в виде разделов библиотек и включать в модули с помощью препроцессорного оператора % INCLUDE. Это предупреждает возникновение ошибок рассогласования и позволяет локализовать изменения таких описаний внутри соответствующих разделов.

4.2.6. Хранение модулей во внешней памяти. На различных этапах разработки программы удобно хранить составляющие ее модули во внешней памяти. В ЕС ЭВМ модули обычно хранятся в библиотечных наборах данных, располагаемых на магнитных дисках.

Модули могут находиться на дисках в трех видах: исходном, объектном и загрузочном. В соответствии с этим организуется 3 библиотеки, каждая из которых в соответствии с требованиями ОС ЕС ЭВМ предназначена для хранения модулей определенного вида. В ходе разработки используются библиотеки всех трех видов.

После препарации модуля он с помощью служебной программы UPDATE (или какой-либо другой) помещается в раздел библиотеки исходных модулей; при этом осуществляется также нумерация строк и распечатка текста модуля. Устранение ошибок препарации из текста модуля производится также с помощью программы UPDATE или в случае, если программа оказалась со множеством ошибок, то она исправляется вручную и записывается в библиотеку заново.

В ходе автономной отладки производится трансляция модуля, находящегося в библиотеке исходных модулей, выполнение (тестирование) его и затем устранение с помощью программы UPDATE ошибок, обнаруженных в тексте отлаживаемой программы.

После того, как модуль прошел автономную отладку, его в объектном (или иногда загрузочном) виде помещают в соответствующую библиотеку. В ходе стыковочной отладки оттранслированные ранее модули с помощью обращения к редактору связей объединяются с другими модулями (транслируемыми или оттранслированными) и выполняются. Если в ходе отладки в каком-либо модуле обнаруживаются ошибки, то они устраняются из текста модуля, хранящегося в исходной библиотеке, и после проверки (отладки) модуля он снова за-

писывается в библиотеку в объектном виде. После того как отлажена какая-то группа модулей, составляющих достаточно самостоятельную часть программы, они с помощью редактора связей могут быть объединены и помещены в библиотеку загрузочных модулей. Это позволит в дальнейшем избежать траты времени на многократные объединения модулей перед выполнением программы. Заметим, что объединение модулей производится редактором связей ОС ЕС как с помощью явного указания объединяемых разделов-модулей, так и путем автоматического подключения модулей по их именам, встречающимся в вызывающих модулях.

Таким образом, в процессе отладки программа хранится во внешней памяти в виде набора разделов, расположенных в одной или нескольких библиотеках: исходной, объектной и загрузочной. При каждом сеансе отладки все модули с помощью средств ОС объединяются в программу, которая выполняется. Некоторые из модулей перед объединением подвергаются необходимому исправлению и трансляции.

Библиотеки для хранения модулей могут быть организованы как для отдельных программистов, так и для целых групп разработчиков, связанных или не связанных между собой. При этом необходимо позаботиться о регулярной чистке библиотеки от ненужных уже вариантов модулей и предусмотреть процедуру восстановления содержимого библиотек на случай сбоя в работе диска.

Аналогичные средства для хранения модулей во внешней памяти имеются и на других ЭВМ.

4.3. Структурированность программы

Помимо модульности другим свойством, которое содействует предупреждению появления в программе логических ошибок, является структурированность. Обычно структурированной называется программа, логическая структура которой отвечает некоторым жестко установленным требованиям. Уже модульную программу можно иногда считать в определенной степени структурированной, поскольку от модульной программы требуется, например, чтобы она состояла только из модулей с одним входом.

4.3.1. Стандартные структуры. Следуя Э. Дейкстре [4], будем считать, что для структурированности программы необходимо, чтобы при ее разработке применялись только следующие (см. рис. 7) логические структуры (схемы).

Приведенным структурам программа должна отвечать как в малом, так и в крупном, т. е. необходимо, чтобы выполнялись следующие условия.

1. Допускаются такие операторы языка, структуры которых соответствуют перечисленным стандартным структурам. Таким образом, в рассматриваемых алгоритмических языках допускаются к использованию только следующие операторы:

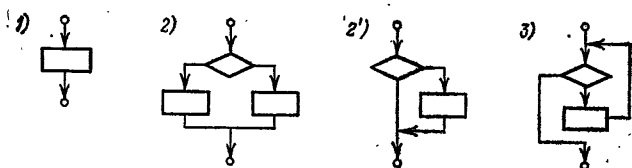


Рис. 7. Стандартные структуры.

1) безусловные операторы, такие, например, как оператор присваивания, оператор вызова процедуры (с возвратом на следующий оператор), пустой оператор, операторы ввода-вывода и т. п.;

2) условный оператор;

3) оператор цикла соответствующего схеме типа.

2. Допустимые операторы могут объединяться или последовательно, или только в соответствии с приведенными стандартными структурами. То есть структурированными будут, например, программы (или фрагменты программ) со следующими схемами (см. рис. 8).

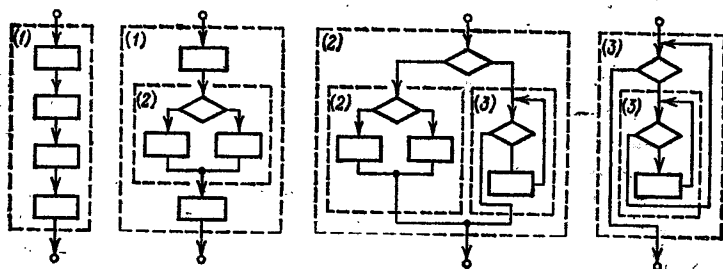


Рис. 8. Стандартное объединение структур.

Пунктиром выделены стандартно объединяемые подструктуры, которые, в свою очередь, отвечают стандартным схемам. В алголе и PL/1 последовательное объединение операторов может осуществляться с помощью составного оператора (DO-оператора) и блока (простого блока в PL/1). Отметим, что как стандартные структуры, так и их объединения имеют только один вход и один выход.

Обращает на себя внимание тот факт, что из широко используемых операторов в рассмотрении отсутствует оператор перехода (GO

ТО). То есть предполагается, что оператор перехода не должен применяться при структурированном программировании. Такое запрещение исходит из того, что использование оператора перехода может в ряде случаев сильно усложнять программу, запутывать ее логику. Здесь необходимо поподробнее разобраться в том, что же следует понимать под простотой программы.

4.3.2. Простота программы. Простой программой предлагается считать такую программу, для которой статическая форма ее представления — текст программы — максимально точно отражает динамический процесс ее выполнения. Или, иными словами, программа считается простой, если последовательность выполнения операторов программы близка к последовательности расположения этих операторов в тексте программы. Действительно, проверка работы такой программы, в особенности, ее логики, очевидно, особых сложностей не вызовет, так как блоки и операторы, составляющие блоки, выполняются, можно сказать, линейно. Изучение и проверка таких программ также происходит линейно — сверху вниз.

Но оператор перехода как раз и предназначен для того, чтобы изменять последовательное выполнение операторов в тексте программы. Именно поэтому он и должен быть запрещен.

Вместе с тем возникает вопрос: можно ли надеяться с помощью только стандартных структур запрограммировать любой алгоритм? Доказано, что можно.

Тогда возникает другой вопрос: стоит ли усложнять процесс программирования, отказываясь от оператора перехода? Многое ли мы получим взамен?

Как уже говорилось, структурированное программирование имеет целью предупредить возникновение логических ошибок в программе, облегчить ее тестирование и отладку. Если не принимать особых мер, то можно считать, что сложность программы с увеличением ее объема растет по квадратичной или кубической зависимости, и в такой же пропорции растет, например, календарное время, необходимое для ее тестирования и отладки. Сложность структурированной программы, ввиду последовательного характера ее выполнения, растет лишь линейно, что для больших программ значительно сокращает время и усилия, необходимые на проведение отладки. И для средних по объему программ упрощение их структуры помогает разработчикам облегчить отладку, особенно тем из них, которые не являются профессиональными программистами и не обладают необходимыми навыками эффективного проведения отладки, требующей больших усилий.

Что же касается трудностей программирования при использовании только ограниченного набора возможностей языка, то здесь можно сделать следующие замечания. Во-первых, оказывается, что

если использовать метод нисходящего проектирования, изложенного ниже (гл. 5), то запрещение оператора перехода становится мало заметным и даже естественным. А, во-вторых, как и для модульности, можно позволить себе иногда снять некоторые ограничения и несколько расширить набор стандартных структур, не очень рискуя при этом запутать логику программы и увеличить трудности последующей отладки.

4.3.3. Дополнительные структуры. Среди дополнительных структур, которые часто используют наряду со стандартными, обычно присутствуют следующие: см. рис. 9.

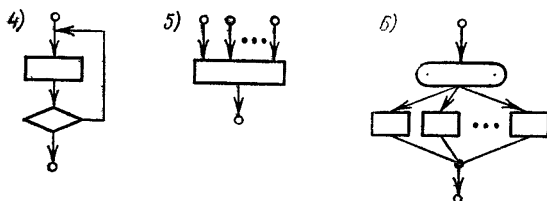


Рис. 9. Дополнительные структуры.

То есть допускается использование следующих возможностей (операторов) в языках:

4) оператор цикла с проверкой исчерпания в конце цикла (как в фортране);

5) несколько входов в блок (оператор ENTRY в фортране и PL/1);

6) переключатель, рассматриваемый как обобщение условного оператора, с требованием, чтобы все альтернирующие блоки имели одну и ту же точку выхода.

Операторов, соответствующих последней схеме, в рассматриваемых языках нет, но ее легко реализовать с помощью переключателя в алголе или, применяя специальные случаи оператора GO TO в PL/1 и фортране (такую реализацию удобно осуществлять с помощью препроцессорных процедур — ср. с п.4.3.4).

Таким образом, использование оператора перехода не запрещается категорически: он может быть использован для реализации некоторых разрешенных структур, для которых в алгоритмическом языке нет соответствующих конструкций. То есть, наличие GO TO в программе еще не говорит о том, что структурированность программы нарушена. С другой стороны, отсутствие оператора перехода в программе не является признаком ее структурированности — программа может быть неструктурированной и по другим причинам (например, программа имеет несколько входов или выходов). Необходимо стремиться разрабатывать программу таким образом, чтобы она естествен-

но получалась структурированной, чтобы, в частности, оператора перехода просто не требовалось, то есть нужно мыслить структурированной! Существенную помощь в освоении такого стиля разработки должна оказать методика нисходящего проектирования (см. гл. 5).

4.3.4. Трудности. Удобство применения структурированного (структурного) программирования зависит как от навыков программиста, так и от возможностей алгоритмического языка, на котором производится программирование. Замечено, что идеи структурированного программирования легче усваивают те, кто имеет дело с алголом или PL/1, языками, в которых четко выражен принцип блочности и имеется возможность трактовать несколько последовательных операторов, как один оператор. Кроме того, легче добиваются структурированности своих программ те программисты, которые, кроме того, имеют опыт работы с макросредствами; и, наоборот, трудности обычно испытывают те, кто пишет программы на фортране и в командах ассемблера.

В фортране, где отсутствуют составной оператор и понятие блочности, трудности вызываются отсутствием вложенности операторов IF и ограниченными возможностями оператора DO. Кроме того, в операторе IF по сравнению с более развитыми языками отсутствует конструкция ELSE, и ее функции могут быть реализованы только с помощью оператора GO TO.

В ассемблере вообще нет составных операторов, и приходится при использовании методов структурированного программирования реализовывать стандартные структуры с помощью имеющихся команд, в частности, и команд безусловного перехода. При этом устанавливают и закрепляют некоторые стандартные приемы (шаблоны) такой реализации, с тем чтобы упростить составление программы и ее дальнейшую проверку.

Использование макросредств позволяет ввести стандартные структуры в язык и облегчить их применение и в ассемблере, и в фортране, и в других алгоритмических языках (для ОС ЕС в двух последних случаях, например, с помощью препроцессорных операторов PL/1 — см. П.5).

Например, в фортране ЕС, используя препроцессорные средства, можно задать следующую последовательность операторов, которая представляет условный оператор, аналогичный имеющемуся в PL/1 и алголе (но без THEN):

```
□IF (A .LT. B)
    B=C
    CALL SUM (A, B, C)
ELSE
    B=A
□ENDIF
```

Идентификаторы `QIF`, `ELSE` и `QENDIF` являются именами препроцессорных процедур, которые после препроцессорной обработки получают, например, следующий вид (в совокупности с заданными выше операторами фортрана):

```
IF (.NOT.(A.LT.B)) GO TO 9021
  B=C
  CALL SUM (A, B, C)
  GO TO 9022
9021 CONTINUE
C    /*ELSE*/
      B=A
  CONTINUE
9022 CONTINUE
C    /*ENDIF*/
```

Сравнение в операторе `IF` является обратным по отношению к заданному выше, которое можно было бы задавать для препроцессора и в обычных обозначениях (`<` вместо `.LT.`). Дополнительные метки `9021` и `9022` генерируются в препроцессорных процедурах `QIF` и `ELSE`. Непомеченный оператор `CONTINUE` пришлось добавить из-за особенностей обращения к препроцессорным процедурам-функциям (см. П.5); по этой же причине пришлось отказаться от ключевого слова `THEN`.

Макрооператор `QIF` может быть вложенным, т. е. стоять на месте любого из операторов фортрана, расположенного до `QENDIF`.

Аналогичным образом можно ввести в фортран составной оператор или, например, цикл с `WHILE`, отсутствие которого остро чувствуется в этом языке. Подробнее использование препроцессорных средств для реализации структурированных схем рассмотрено в книге Дж. Хьюза и Дж. Мичтома [7], но там имеется ввиду специальный препроцессор для языка фортран.

Возвращаясь к изложению основного материала, заметим, что другими нестандартными возможностями, которые иногда используют, отходя от канонов структурированности, являются следующие:

- 7) выход из цикла по `GO TO`;
- 8) второй выход из модуля, для случая возникновения аварийной ситуации в нем;
- 9) переход по `GO TO` вперед для модуля, размещающегося на одном листе бумаги.

Кроме того, явно неструктурированным средством, которое часто приходится использовать, является `ON`-оператор в `PL/1`: он может быть включен в выполнение при возникновении исключительной ситуации в любом месте программы, и поэтому естественная после-

довательность выполнения операторов программы в любой момент может быть нарушена.

Использование неканонических структур должно допускаться лишь после их опробования, освоения и утверждения для некоторой группы разработчиков с тем, чтобы предотвратить возникновение ошибок при использовании таких структур и позволить легко распознавать их при проверке программ (ср. со стандартами программирования в 4.1.5).

Эффективность отдельных структурированных блоков программы из-за отказа от ряда полезных в некотором конкретном случае неструктурированных возможностей языка может оказаться ниже (на 5—10%) по сравнению с обычными неструктурированными блоками: это является платой за простоту тестирования и легкость локализации ошибок, за уменьшение количества ошибок в составленной программе. Кроме того, можно сказать, что такая потеря эффективности является локальной: в силу простоты логической структуры, лучшего представления программиста о действительной структуре его программы и лучшей ее продуманности можно ожидать, что все это даст повышение глобальной эффективности структурированной программы.

ГЛАВА 5

НИСХОДЯЩЕЕ ПРОЕКТИРОВАНИЕ

В предыдущей главе было выдвинуто много требований, которым должна удовлетворять составляемая программа для того, чтобы ее можно было легко отладить. Но требований было высказано столько, что у читателя, должно быть, сложилось впечатление, что если пытаться их все удовлетворить, то программирование станет слишком трудным занятием. Нельзя ли эти требования сформулировать в более общей форме или предложить такую методику разработки программ, при следовании которой программы получались бы простыми и наглядными, модульными и структурированными? Такой метод существует — он называется нисходящим проектированием или проектированием сверху — вниз, а также — иерархическим, пошаговым или систематическим программированием (проектированием).

Собственно говоря, этот метод не так уж нов, можно считать что он является развитием такого приема при разработке сложных программ, когда составляют несколько уровней блок-схем, от самой общей до самой детальной, и на каждом уровне производится детализация алгоритма.

5.1. Общие положения

5.1.1. Сущность нисходящего проектирования. Под нисходящим проектированием понимается некоторая совокупность методов эффективной разработки (и отладки) программ. Сущность нисходящего проектирования заключается в том, что проектирование и алгоритмизация программы производятся мелкими шагами, на каждом из которых программист принимает как можно меньше алгоритмизирующих решений, обычно одно; причем каждый шаг является шагом в направлении все большей конкретизации алгоритма. Конкретизация может идти неравномерно: одни части могут быть уже доведены до уровня языка программирования, а для других алгоритмизация только начинается,

Впрочем, есть и другие представления о сущности нисходящего проектирования: некоторые считают, что основной его идеей является требование ясности мышления в ходе проектирования программы, то есть максимально четкое представление о поставленной задаче, о путях ее решения и о последствиях принимаемых решений.

Первое знакомство с нисходящим проектированием лучше привести на примерах. Приводимые ниже примеры взяты из работы Э. Дейкстры [4]. Заимствование этих примеров можно объяснить тем, что так как программирование является, по сути дела, искусством, то учиться ему нужно на образцах, созданных выдающимися программистами. Но поскольку настоящая книга имеет более практические цели чем работа Э. Дейкстры, примеры заново и более подробно прокомментированы (конечно, с некоторым риском исказить при этом действительные мысли автора) и доведены до программ на языке PL/I; кроме того, эти программы даются также на фортране или алголе-60.

Приводимые решения задач следует рассматривать как иллюстрацию самого общего подхода к разработке программ, а не как образец программирования типовых задач в данной конкретной области; для каждого типа задач должны быть, кроме того, разработаны свои, свойственные только им приемы проектирования в рамках общей методики.

5.1.2. Первый пример. Требуется ввести текст, состоящий из слов, разделенных одним или несколькими пробелами, и оканчивающийся точкой (перед ней могут быть и пробелы), преобразовать его и вывести. Преобразование заключается в том, что слова разделяются только одним пробелом, точка ставится сразу за последним словом и символы всех слов с четными номерами должны выводиться в обратном порядке. Ввод осуществляется с использованием функции INSYM (без параметра), получающей значение очередного вводимого символа; вывод производится процедурой (подпрограммой) OUTSYM (x), осуществляющей печать символа x в очередной позиции. Таким образом, и при вводе и при печати возврат на предыдущие позиции невозможен. Длина слов не превышает 20 символов. Например, текст

эта программа никуда не годится,

должен быть преобразован в следующий:

эта маргорт никуда не годится.

В этом месте читатель должен отложить в сторону книгу и решить предложенную задачу самостоятельно; это позволит ему значительно лучше разобраться в предлагаемом ниже решении. Только сравнивая свой путь решения задачи с методом Э. Дейкстры, можно глубоко понять его подход к нисходящему проектированию.

В дальнейшем при решении задачи на первых шагах детализации будет использоваться вспомогательный алгоритмический язык, в котором, в основном, используются элементы таких наиболее гибких и распространенных языков, как алгол-60 и PL/1; обычно каждый программист выбирает вспомогательный язык по своему вкусу (подробнее об этом см. в 5.2.3).

1°. Для начала попытаемся представить себе самую общую структуру составляемой программы. Первую структуру, которая, должно быть, приходит в голову, можно выразить следующей простейшей последовательностью основных этапов обработки текста:

преобразование;

КОНЕЦ

Очевидно, для ограничения требований к объему памяти необходимо проводить обработку текста какими-то порциями, выполняя сразу их ввод, преобразование и вывод. То есть, программа должна иметь следующее строение:

ЦИКЛ

КОНЕЦ

1в, программа ~ начало увертюра;

обработка порции;

финал

КОНЕЦ

2°. Теперь нужно заняться детализацией этапа обработки порции, который требуется разбить на более мелкие, более конкретные подэтапы. Например, можно считать, что обработку порции естественно представить в виде следующих трех подэтапов:

2а. обработка порции ~

~ начало ввод порции;
 преобразование;
 вывод преобразованной порции

конец

Преобразование, в свою очередь, можно разделить на сокращение количества пробелов и перевертывание слов (через одно слово).

Но все эти рассуждения носят, все-таки, чересчур абстрактный характер: они не учитывают конкретного вида порции, с которой придется иметь дело. Может случиться, что требуемое преобразование или его часть окажется необходимым или возможным совместить как-то со вводом или выводом. Например, обработка порции может иметь вид:

ввод + преобразование1; преобразование2 + вывод

или

ввод + преобразование1; преобразование2; вывод

Никаких конкретных критериев для того, чтобы выбрать один из возможных способов разбиения процесса обработки очередной порции на подэтапы, у нас пока нет. Поэтому следует признать, что такой выбор делать преждевременно до тех пор, пока не конкретизировано понятие порции. Нам придется обратиться к рассмотрению обрабатываемого текста и попытаться решить вопрос о его разбиении на порции.

Видимо, можно рассматривать только такую порцию, которая содержит одно слово вместе с соседними пробелами (или точкой). Другие варианты, когда в качестве порции берется несколько слов или символов, представляются менее естественными (а для символов — и, вообще, неподходящими из-за характера требуемого преобразования).

Исходя из цикличности обработки текста и, тем самым, из цикличности разбиения его на порции, каждая из порций должна содержать помимо слова также и соседние пробелы (предшествующие, или последующие, или часть тех и других). Действительно, если, например, рассмотреть текст, приведенный выше, то где бы ни ставить, ориентируясь на какое-нибудь простое правило (на регулярный циклический процесс обработки), границы между порциями, содержащими целое слово, пробелы, разделяющие слова, должны войти в одну из соседствующих порций. Вспомним, однако, что в формулировке задачи ограничение на количество пробелов между словами отсутствует, и поэтому максимальная длина выбранной порции оказывается неопределенной. Как же можно обрабатывать такие порции? Где же выход?

Но ведь обработка той части порции, которая состоит из пробелов, заключается лишь в выбрасывании всех (может быть, кроме одного) пробелов! Таким образом, оказывается, что выбрасывание пробелов необходимо делать непосредственно при вводе, без запоминания их в памяти программы.

Печать с переворачиванием слова без его запоминания не сделаешь, и поэтому символы слова необходимо сохранять при вводе. Что же касается самого преобразования, то оно имеет достаточно простой характер, и, видимо, можно уже сейчас представить себе, как выполнить его вместе с выводом, хотя в данном случае это и не принципиально (если возникнут какие-либо трудности при дальнейшей разработке, то можно переворачивание слова выполнить до начала вывода).

Итак, принятие решения о том, что порция содержит одно слово текста (с разделяющими пробелами), позволило нам следующим образом конкретизировать этап обработки порции:

26. обработка порции ~

~ начало ввод порции с пропуском пробелов;
 вывод слова с переворачиванием через раз
 конец

Из приведенного рассуждения следует также, что в дальнейшем нужно различать три вида порций: вводимая (с пробелами), преобразуемая (без пробелов) и выведенная (может быть перевернутая).

3°. Для дальнейшей конкретизации алгоритма обработки порции следует, видимо, уточнить строение различных порций, решить, например, вопрос о месте пробелов во вводимой порции: будут ли они в начале порции или в ее конце, или и там и там? Например, приведенный выше пример текста можно разбить на порции различными способами (не обращайтесь пока внимания на крайние порции):

- 1) эта|_|_|_|программа|_|никуда|_|не|_|_|годится|_|_|.
- 2) эта|_|_|_|программа|_|никуда|_|_|не|_|_|_|годится|_|_|.
- 3) эта|_|_|_|программа|_|никуда|_|_|не|_|_|_|годится|_|_|_|.

Вспомним, однако, о том, что функция INSYM вводит символы последовательно один за другим, и поэтому нельзя узнать, кончился ли уже ввод слова или нет, кроме как путем ввода следующего за ним пробела. А об окончании ввода последнего пробела между словами можно узнать, только введя первую букву следующего слова. Поэтому первый вариант разбиения на порции оказывается невозможным, и вместо него следует рассматривать 3-й вариант; для второго варианта порции ее необходимо видоизменить, поскольку она должна оканчиваться первой буквой следующего слова.

Таким образом, в дальнейшем можно рассматривать только следующие два вида вводимых порций:

1) слово с одним пробелом или точкой после него и, может быть, с несколькими пробелами перед ним;

2) слово без первой буквы со всеми имеющимися пробелами после него и первой буквой следующего слова (или с точкой — для последней порции).

Поэтому наш пример текста может быть разделен на порции следующими двумя способами:

1) |эта|_|_|программа|_|никуда|_|не|_|_|годится|_|_|.

2) |эта|_|_|_|_|п|_|рограмма|_|_|и|_|куда|_|_|_|_|не|_|_|_|_|е|_|_|_|_|е|_|_|_|_|годится|_|_|_|_|.

Нам предстоит выбрать один из двух видов порций для дальнейшей работы. Рассмотрим их повнимательнее и попытаемся представить себе, для какого вида порции программа должна получиться проще. В первом случае последняя порция имеет особый вид; а во втором — первая буква текста выпадает из цикла, и порция кажется не вполне естественной, так как в ней присутствуют части двух соседних слов. Общая структура программы, видимо, будет проще все-таки для 2-го случая, поскольку цикличности обработки текста легко добиться, введя первую букву текста перед циклом. Для первого вида порции ввод хвостовых пробелов и точки в финале программы сделать было бы, пожалуй, несколько труднее.

Помимо рассмотрения структуры вводимой порции следует продумать и строение порции выводимого текста. Но здесь дело обстоит проще, так как вполне естественно рассматривать лишь один вид порции, содержащей выводимое слово, сопровождаемое пробелом (или точкой — для последнего слова). Например, в приведенном выше примере выведенный текст разбивается на порции следующим образом:

|эта|_|_|аммаргорт|_|_|никуда|_|_|не|_|_|годится|_|_|.

Заметим в скобках, что преобразуемые порции имели следующий вид:

эта|программа|никуда|не|годится|

Очевидно, чем больше вводимые и выводимые порции будут соответствовать друг другу, тем легче будет осуществить преобразование и вывод. Кажется, что это соответствие лучше для 1-го варианта вводимой порции, поскольку во втором варианте порция не содержит целого слова. Но, с другой стороны, в 1-м варианте последняя порция может не содержать точки, что нарушает соответствие вводимой и выводимой порций, и является серьезным доводом в пользу порций 2-го вида, несмотря на их внешнюю «неестественность».

Итак, мы принимаем решение: ориентироваться при разработке алгоритма на порции 2-го вида. Нужно, однако, быть готовым к тому, что некоторая неестественность порции может все же вызвать логические затруднения при реализации алгоритма.

Теперь алгоритм можно детализировать следующим образом:

программа~

~начало увертюра; ввод символа;

пока (не точка) цикл

начало ввод порции с пропуском пробелов;

вывод порции с перевертыванием через раз

конец;

финал

конец

4°. Приступая к дальнейшей конкретизации разрабатываемого алгоритма, рассмотрим повнимательнее структуру вводимой порции. Можно заметить, что все пробелы (пробел) в порции сосредоточены в конце порции, где, кроме того, находится и первый символ следующего слова. Вводимые символы слова требуется запомнить, пробелы пропустить, а последний символ порции, если он не точка, сохранить до ввода следующего слова, к которому сохраненный символ и нужно будет добавить в начале обработки следующей порции. Вот такая вырисовывается последовательность обработки вводимой порции. Что касается вывода, то совмещение его с перевертыванием, если оно потребуется, можно достигнуть просто, производя вывод слова в обратном порядке,

Зафиксируем алгоритмы ввода и вывода, вытекающие из принятых решений по структуре порции:

ввод порции~начало

запоминание 1-го символа слова;

ввод слова с запоминанием;

пропуск пробелов с сохранением символа

следующего слова

конец

вывод порции~начало

вывод слова с конца или с начала;

вывод пробела или точки

конец

5°. Алгоритм уже достаточно конкретизирован, и можно было бы попытаться выразить его на каком-либо языке высокого уровня, например, на PL/1. Но лучше выполнить еще один шаг детализации и всюду довести алгоритм до действий над символами, так как могут быть еще всякие неожиданности.

6°. Прежде чем писать операторы PL/1 нужно решить, как будут представлены на этом языке те величины, которые используются в нашем алгоритме.

Введенное слово можно запомнить или в символьной строке или в символьном массиве. Последний вариант, видимо, предпочтительнее для работы с отдельными символами слова, и поэтому следует остановиться на нем. Для анализа введенного символа и для сохранения его до следующего выполнения цикла необходима символьная переменная. Таким образом, задаем следующее описание

DECLARE WORD (1 : 20) CHAR (1), SYM CHAR (1);

Если воспользоваться понятием структуры в PL/1, то для лучшего отображения строения переменной, представляющей слово, следовало бы задать:

1 WORD, 2 BUKWA(1:20) CHAR(1)

и везде далее писать WORD.BUKWA(1) или просто BUKWA(1).

Конкретизация операторов:

ввод символа ~ SYM=INSYM;

ввод слова ~ DO I=1 TO 20 WHILE (SYM \neg =' ' & SYM \neg ='.');

BUKWA (I)=SYM;

SYM=INSYM;

END;

пропускание пробелов ~ DO WHILE(SYM=' ');

SYM=INSYM; END;

Для того, чтобы управлять выводом слова то в прямом, то в обратном направлении, можно предложить несколько способов:

1) подсчитывать номер выводимого (вводимого) слова и определять его четность;

2) умножать какую-либо арифметическую переменную каждый раз на -1 и определять знак значения переменной;

3) задать логическую (битовую) переменную и для разветвления менять ее значение на обратное операцией логического отрицания (\neg) после вывода каждого слова.

Последние два способа являются более простыми чем первый, а из них 3-й способ — более экономичным. Поэтому вне цикла (в увертюре) следует присвоить логической переменной исходное значение, а в цикле менять значение этой переменной на противоположное.

Итак:

увертюра ~ DECLARE DIR BIT (1) INITIAL ('1'B);

изменение направления печати ~ DIR= \neg DIR;

вывод слова ~ IF DIR THEN *печать слова с начала*
ELSE *печать слова с конца*

вывод пробела или точки ~

```
~ IF SYM='.' THEN CALL OUTSYM ('.');
  ELSE CALL OUTSYM (' ');
```

Вывод пробела или точки производится в зависимости от того, какой символ (отличный от пробела) был введен в конце очередной порции вслед за пропущенными пробелами,

печатать слова с начала ~

```
~DO K=1 BY 1 TO I;
```

```
  CALL OUTSYM (BUKWA(K)); END;
```

печатать слова с конца ~

```
~DO K=I BY-1 TO 1;
```

```
  CALL OUTSYM (BUKWA(K)); END;
```

7°. Теперь нужно собрать программу на PL/I из полученных ранее фрагментов. Направляется составление программы из трех модулей: ввода порции, вывода порции и из центрального модуля, содержащего увертюру и обращения к двум другим модулям, вызывающим в свою очередь процедуры-модули INSYM и OUTSYM (см. рис. 10).

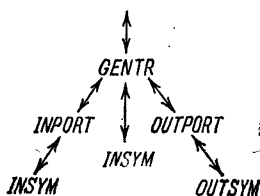


Рис. 10. Схема программы для примера 1.

Но поскольку разработанные модули состоят всего лишь из нескольких операторов, практически имеет смысл объединить первые 3 модуля в один; кроме того, ради простоты оформим наш модуль в виде блока (полная программа приведена в 6.1).

```

/*****
/* ПРЕОБРАЗОВАНИЕ ТЕКСТА ДЛЯ ПОРЦИЙ ТИПА 2 */
/* ПЕРЕВЕРТЫВАНИЕ ЧЕТНЫХ СЛОВ
/* И СОКРАЩЕНИЕ ПРОБЕЛОВ
/* НАЧАЛО ВВОД СИМВОЛА;
/* ПОКА НЕ ТОЧКА ЦИКЛ ПО ПОРЦИЯМ
/* ВВОД СЛОВА;
/* ПРОПУСКАНИЕ ПРОБЕЛОВ;
/* ВЫВОД СЛОВА С КОНЦА ИЛИ С НАЧАЛА;
/* ВЫВОД ПРОБЕЛА ИЛИ ТОЧКИ
/* КОНЕЦ ЦИКЛА ПО ПОРЦИЯМ
/* КОНЕЦ
*****/
```

```
TRANSF: BEGIN;
```

```
  DECLARE 1 WORD,
```

```
    2 BUKWA (1:20) CHAR (1) INIT ((20) '#'),
```

```

SYM CHAR (1) INITIAL ('@');
DIR BIT (1) INITIAL ('1'B);
(I, K) FIXED BINARY;
□□: SYM=INSYM; /* ВВОД СИМВОЛА В УВЕРТЮРЕ */
      /* ЦИКЛ ПО ПОРЦИЯМ */
CYCL: DO WHILE (SYM≠'.');
      IN: DO I=1 TO 20
            WHILE (SYM≠'␣' & SYM≠'.');
            BUKWA (I)=SYM; SYM=INSYM;
            END IN; /* ВВОД СЛОВА */
      MISS: DO WHILE (SYM='␣');
            SYM=INSYM;
            END MISS; /* ПРОПУСК ПРОБЕЛОВ */
            /* ВЫВОД СЛОВА */
      OUT: IF DIR
            THEN DO K=1 BY 1 TO I
                  CALL OUTSYM (BUKWA (K));
                  END;
            ELSE DO K=I BY -1 TO 1;
                  CALL OUTSYM (BUKWA (K));
                  END;
            IF SYM≠'.' /* ВЫВОД */
            THEN CALL OUTSYM ('␣'); /* РАЗДЕ- */
            ELSE CALL OUTSYM ('. '); /* ЛИТЕЛЯ */
            DIR=¬DIR; /* ЧЕТ-НЕЧЕТ */
      END CYCL;
END TRANSF;

```

Инициализация переменных BUKWA и SYM производится в целях предстоящей отладки. Как и каждая настоящая вновь составленная программа, эта тоже содержит ошибки, и читателям предстоит их найти за столом или с использованием машины (см. 6.1).

Полученная программа, видимо, не является самой краткой из возможных, поскольку, например, содержит 3 пробы на точку, что является явным излишеством. Но программе нельзя отказать в четкости структуры, в ясности функционального смысла каждого оператора, в соответствии порядка операторов в тексте программы и последовательности их выполнения.

Конечно, приведенные выше рассуждения выглядят слишком многословными, а характер разработки мало похож на практическую работу программиста при решении задачи. Но здесь фиксировались все мысли и соображения, которые приходили (или могли

приходить) по ходу решения задачи. На практике такая подробная фиксация, конечно, не нужна: необходимо только представлять на бумаге получаемые промежуточные алгоритмы и строение фигурирующих в них величин с комментариями. Поэтому, на самом деле, большинство приведенных размышлений, которые отнимают так много времени при чтении, в голове хорошего программиста могут проноситься почти мгновенно.

6°. Здесь дается решение предложенной задачи на фортране-IV. Нам придется вернуться к п. 5° и рассмотреть вопрос о представлении на языке фортран данных, используемых в разработанном алгоритме.

В фортране нет символьных переменных, но ввод символьной информации (и вывод ее) можно осуществить, используя формат типа A. В качестве переменных при этом могут быть взяты переменные любого вида, но наиболее экономичными представляются логические с длиной, равной единице (для транслятора уровня H), или — целые с длиной 2 (для уровня G). Описание величин, которое годится для обоих трансляторов, получает следующий вид:

INTEGER*2 WORD (20), SYM

Теперь заново нужно провести конкретизацию операторов, определенных в п. 5°. Ниже используется один из возможных способов перехода от операторов использованного условного алгоритмического языка к неструктурированным операторам фортрана, в котором невозможно избежать применения «запрещенного» оператора GO TO.

Конкретизация операторов п. 5°:

ввод символа ~ SYM=INSYM (F)

Поскольку в фортране не допускается функция без параметров, приходится добавить фиктивный (пустой) аргумент и считать, что INSYM имеет один параметр; имя F имеет смысл добавить к предыдущему описанию.

ввод слова ~

L=0

DO 10 I=INF, INI, TE

IF (SYM.EQ.BLANK.OR.SYM.EQ.POINT) GO TO 20

WORD (I)=SYM

SYM=INSYM (F)

L=L+1

10 CONTINUE

20

Целые переменные BLANK и POINT, имеющие значения ' ' и '.', использованы поскольку символьные константы не могут яв-

ляться операндами операций сравнения. Целая переменная L введена для вычисления длины вводимого слова, поскольку параметр цикла в фортране (в отличие от PL/1) в случае окончания цикла по истечению (то есть при длине слова равном 20) имеет неопределенное значение.

пропуск пробелов ~

```
DO 30 J=INF, INI, TE
  IF (SYM.NE.BLANK) GO TO 40
  SYM=INSYM (F)
30 CONTINUE
40 . . . . .
```

Первые два оператора каждого из приведенных циклов можно рассматривать как «заголовок» цикла, в котором дается полная характеристика задаваемого циклического процесса: первый оператор определяет максимальную кратность повторения цикла и правило изменения параметра цикла, а второй — дополнительное условие окончания цикла. Переменные INF, INI, TE в операторе DO использованы для задания бесконечного цикла; присваивание им необходимых значений можно провести при их описании (шаг, равный нулю, в фортране ЕС является допустимым):

INTEGER INF/1/, INI/2/, TE/0/

Оператор CONTINUE применяется для единообразия в реализации и для идентификации конца цикла. Используя такие же или подобные схемы программирования, можно конкретизировать и остальные операторы п. 5°, что читателю и предлагается проделать самостоятельно.

Ниже приводится уже полный алгоритм, оформленный в виде подпрограммы фортрана (см. пример). Комментарии в примере оформлены по правилам П. 7, для возможности использования в дальнейшем препроцессорных средств.

Пример на фортране.

SUBROUTINE TRANSF

```
G  /* ПРЕОБРАЗОВАНИЕ ТЕКСТА */
    INTEGER*2 WORD (20)/20'##', SYM/'##'/
    INTEGER*2 POINT/'.'/, BLANK/'_'/, F, INSYM
    INTEGER I, J, K, L, JP
    INTEGER INF/1/, INI/2/, TE/0/
    LOGICAL*1 DIR
G  /* УВЕДОМЛЕНИЕ */
00001 DIR=.TRUE.
    SYM=INSYM (F)
```

```

C  /* ЦИКЛ ПО ПОРЦИЯМ */
05      DO 110 JP=INF, INI, TE
        IF (SYM .EQ. POINT) GO TO 120
G  /* ВВОД СЛОВА */
        L=0
        DO 10 I=1, 20, 1
        IF (SYM .EQ. BLANK .OR. SYM .EQ. POINT) GO TO 20
        WORD(I)=SYM
        SYM=INSYM (F)
08      L=L+1
10      CONTINUE
C  /* ПРОПУСК ПРОБЕЛОВ */
20      DO 30 J=INF, INI, TE
        IF (SYM .NE. BLANK) GO TO 40
        SYM=INSYM (F)
30      CONTINUE
C  /* ВЫВОД ПРЯМО ИЛИ ОБРАТНО */
40      IF (.NOT. DIR) GO TO 60
        DO 50 K=1, L, 1
        CALL OUTSYM (WORD(K))
50      CONTINUE
        GO TO 80
60      DO 70 K=1, L, 1
        KB=L-K+1
        CALL OUTSYM (WORD(KB))
70      CONTINUE
C  /* ПЕЧАТЬ РАЗДЕЛИТЕЛЯ */
80      IF (SYM .EQ. POINT) GO TO 90
        CALL OUTSYM (BLANK)
        GO TO 100
90      CALL OUTSYM (POINT)
100     DIR=.NOT. DIR
110     CONTINUE
C  /* КОНЕЦ ЦИКЛА ПО ПОРЦИЯМ */
120     RETURN
      END

```

Теперь, после разбора примера можно более детально изложить основные идеи нисходящего проектирования.

5.1.3. Очевидность решений. Алгоритм решения предложенной задачи получается в несколько шагов (в примере 6 шагов), на каждом из которых он все более детализируется. Детализация проводится мелкими шагами, что имеет целью облегчить разработку алгорит-

ма и, тем самым, предупредить возникновение логических ошибок. Мелкие шаги позволяют добиться и того, чтобы принимались только такие решения, которые являются очевидными или наиболее естественными для поставленной задачи.

Критерий естественности принимаемых решений зависит от имеющегося у разработчика опыта алгоритмизации подобных задач, от выработанных и стандартизованных приемов, а также основывается на тщательном анализе постановки решаемой задачи, на максимально конкретном и ясном представлении о характере реализуемого вычислительного процесса. То есть под естественностью и очевидностью решения следует, скорее, понимать его предельное соответствие существу поставленной задачи.

В случае, когда имеются сомнения в правильности принимаемых решений или отсутствуют достаточные основания для выбора одного из нескольких решений, следует, если возможно, отложить принятие таких решений на другие, более поздние шаги алгоритмизации, а пока обратиться к другим сторонам или частям реализуемого алгоритма.

Конечно, и «естественные» решения могут оказаться ошибочными, и придется в ходе разработки программы отменять ранее принятое решение и возвращаться к более ранней стадии разработки. Так что, в каждом решении есть свой риск и его нужно, по возможности, свести к минимуму, сразу подвергнув алгоритм, вытекающий из принятого решения, критической оценке. Можно предполагать, что минимальность риска достигается минимальностью шага алгоритмизации, естественностью и простотой решения. Как говорит Э. Дейкстра, мастерство программиста в том и состоит, что он старается найти «самое легкое решение, то есть такое решение, которое требовало бы минимальных усилий при максимально обоснованной надежде на то, что не придется сожалеть об этом решении». Искусство программиста заключается в умении предвидеть последствия принимаемых решений, и нужно осваивать это искусство, стремиться развивать в себе такую способность.

5.1.4. Обобщенные данные. Одной из основных идей нисходящего проектирования является использование на ранних стадиях алгоритмизации обобщенных, абстрактных данных. В примере такими являются порция и ее части (слово + группа пробелов). На начальных этапах программист имеет дело с крупными частями алгоритма и с крупными данными (текстом, порциями). Далее наряду с детализацией частей алгоритма, детализируются и данные: они разбиваются на более мелкие части (части порций, символы). Таким образом, наряду с вложенностью, иерархичностью программных элементов, можно говорить и об иерархичности элементов данных, над которыми производят действия программные блоки, И как при нис-

ходящем проектировании необходимо было следить за структурированностью программы, точно так же необходимо, чтобы и данные были структурированы, чтобы данные более высокого уровня наиболее просто выражались через элементы данных более низкого, более конкретного уровня. В примере текст состоит из порций, порция — из слова и группы разделителей, слово — из символов.

Две стороны детализации — блоков и данных — тесно связаны друг с другом: одна определяет другую или следует из нее. Например, решение о цикличности обработки текста следует из его неограниченности; необходимость введения понятия порции следует из требования цикличности обработки текста; последовательность обработки каждой порции следует из принятого решения по ее структуре и т. п. Таким образом, если трудно принять какое-либо решение по структуре программы, то необходимо обратить свое внимание на структуру обрабатываемых данных, а при разработке структуры данных нужно исходить из заданных или выявленных свойств разрабатываемого алгоритма. Причем, главными находками, оказывающими решающее влияние на успех разработки, являются именно решения, определяющие выбор структур данных. Именно от выбора основных величин, в терминах которых будет формулироваться алгоритм решения, будет зависеть простота структуры программы, а, следовательно, и простота ее проверки и отладки (см., например, п. 5.3.2). «Представление данных — это сущность программирования» — так формулирует Ф. Брукс [1] решающую роль выбора структуры основных данных при алгоритмизации.

5.2. Характерные черты

Перед обсуждением других свойств нисходящего проектирования рассмотрим еще один пример.

5.2.1. Второй пример. Имеется кривая, заданная параметрически целочисленными функциями f и g :

$$x=f(i), 0 \leq x \leq 99;$$

$$y=g(i), 0 \leq y \leq 49;$$

для $i=1, 2, \dots, 1000$. Требуется напечатать график этой функции, используя следующие два оператора печати:

NEXT (n) — печать пробела (для $n=1$) или звездочки (для $n=2$) в очередной позиции и подвод следующей позиции;

MOVE — перевод строки и подвод 1-й позиции следующей строки печати.

Для определенности будем считать, что ось x направлена вдоль строки печати. Необходимо обратить внимание на то, чтобы в программе выполнялось минимальное количество операторов печати.

Здесь, как и при разборе предыдущего примера, читатель должен отложить в сторону книгу и перед тем, как продолжить чтение, решить задачу самостоятельно. Для тех читателей, которые не имеют практики работы с параметрическими функциями, рекомендуется сначала попрактиковаться в задании простейших функций, отвечающих перечисленным выше условиям. Для того, чтобы при работе иметь перед глазами какую-либо кривую, зададим, например, следующие (см. рис. 11):

$$\begin{aligned}
 &1) \quad x1=30; \quad y1=\text{entier}(i/25); \\
 &2) \quad x2=\begin{cases} 2(i-1) & \text{для } i=1-50; \\ 2(i-51) & \text{для } i=51-100; \\ 99 & \text{для } i > 100. \end{cases} \\
 &y2=\begin{cases} i-1 & \text{для } i=1-50; \\ 100-i & \text{для } i=51-100; \\ 25 & \text{для } i > 100. \end{cases}
 \end{aligned}$$

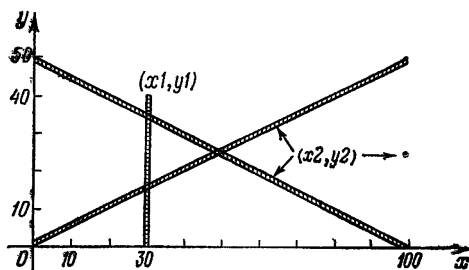


Рис. 11. Графики простейших функций.

1°. В качестве отправной точки для дальнейшей детализации алгоритма можно взять следующую программу, сомнений в правильности которой не возникает:

программа ~ начало нарисовать кривую конец

Наиболее простое решение, которое может быть предложено в качестве первого шага детализации алгоритма, а именно:

1а. *нарисовать кривую ~*

~ для i=1 до 1000 цикл печать '' в точке (f(i), g(i))*

не годится из-за ограниченных возможностей заданных операторов печати. Действительно, операторы NEXT и MOVE позволяют осуществить вывод на печать только слева направо (по 100 позициям в строке) и сверху вниз (по 50 строчкам). Поэтому в начале разработки алгоритма попытаемся удовлетворить в первую очередь требованиям операторов печати, являющихся в нашем случае, так сказать, узким местом. Тогда появляется следующее решение:

16. нарисовать кривую ~

~ пока не все точки

цикл печать '*' или '□' в очередной точке

Остается только решить, как управлять печатью вида символа (* или □). Для каждой из 5000 точек с координатами (x° , y°) нужно узнать, есть ли среди 1000 пар ($f(i)$, $g(i)$) такая, для которой $x^\circ = f(i^\circ)$ и $y^\circ = g(i^\circ)$. Производить всякий раз, для каждой из 5000 точек нашего поля печати вычисление f и g , конечно же, неразумно, и поэтому их нужно вычислить заранее и запомнить.

Для экономии времени поиска, вообще, следовало бы отсортировать хранящиеся пары координат так, чтобы их расположение соответствовало последовательности печати, т. е. по строчкам (по уменьшению игреков), а внутри строчек по позициям (по увеличению иксов). Тогда не нужно будет ничего искать, а просто вид печатаемого символа (* или □) определяется по текущей паре координат: совпадение координат печатаемой точки с текущей парой вычисленных координат определяет печать звездочки, несовпадение — печать пробела.

Мы получили приемлемый проект алгоритма, но у него есть недостатки: необходимость отвести место для хранения 1000 пар координат кривой и, главное, необходимость сортировки пар. Если бы удалось как-то обойтись без сортировки и сохранить простоту структуры рассматриваемого алгоритма, то задачу можно было бы считать решенной успешно. Для этого нужно, чтобы информация о виде печатаемого символа для каждой из 5000 точек могла быть без сортировки представлена в такой форме, которая позволяла бы достаточно просто определять печатать ли звездочку или пробел. Может быть нам удастся решить эту проблему, может быть нет, но у нас в запасе есть предложенное выше решение. Пока мы будем считать, что имеется некоторая величина, значение которой позволяет легко определить вид печатаемого символа для каждой точки бланка, содержащего 50 строчек по 100 позиций в каждой. То есть эта величина каким-то способом отражает вид печатаемой на бумаге кривой, и поэтому будем называть ее *отражением*. Очевидно, отражение является примером одной из абстрактных, обобщенных величин, использование которых необходимо при нисходящем проектировании, поэтому можно считать, что мы на правильном пути. Каким образом отражение соотносится с видами величин, имеющимися в конкретных алгоритмических языках (массивы, структуры, строки и т. п.), мы пока не задумываемся — этим нам предстоит заниматься в дальнейшем. А пока попытаемся уточнить структуру введенной величины, обратившись к рассмотрению алгоритма, чтобы выявить те требования, которые он предъявляет к отражению.

Итак, мы установили, что прежде, чем печатать график, нужно построить его отражение, то есть:

1в, нарисовать кривую ~ {построить (отражение);
напечатать (отражение)}

Здесь используются несколько иные языковые средства при реализации нисходящего проектирования. Отражение, заключенное в круглые скобки, является как бы аргументом соответствующих процедур. Фигурные скобки используются для более четкого указания границ детализирующей группы операторов (или описаний — в дальнейшем); можно считать, что здесь они заменяют границы составного оператора алгола или PL/1.

Конечно, построить отражение, не представляя себе конкретно самого отражения, довольно трудно. Но если пока отвлечься от средств и понятий алгоритмических языков и считать, что свойства отражения и действия над ним максимально отображают свойства реальной кривой на бумаге, то естественно предположить, что построение отражения состоит в расстановке звездочек в нужных местах. Причем, опять же естественно считать, что установка звездочки в то место отражения, где она уже стоит, не меняет вида отражения (как не меняла бы вида кривой и реальная печать звездочки на бумаге в заполненную уже позицию).

Но тогда, продолжив аналогию, нужно предположить, что отражение, перед тем как на нем строить кривую, должно быть чистым, т. е. содержать только пробелы и ни одной звездочки. Таким образом, получаем:

построить ~ {очистить (отражение);
поставить звездочки (отражение)}

Звездочки (а, может быть, и какие-либо другие знаки для других значений n оператора NEXT) ставятся в точку отражения с координатами x и y , значения которых определяются вычислением функций f и g . Таким образом:

```

поставить звездочки ~ DO I=1 BY 1 TO 1000;
                        добавить знак (отражение, 1)
END;

```

добавить знак ~ {DECLARE (X, Y) FIXED (2);
X=F(I); Y=G(I);
поставить знак (отражение; X, Y))

Здесь вместо операторов условного языка использованы операторы PL/1; пример покажет, что применение операторов конкретного алгоритмического языка высокого уровня, каким является PL/1, имеет и свои преимущества. Величины I , X , Y , помещенные в скоб-

ки, являются аргументами соответствующих условных (обобщенных) процедур. Описание X и Y определяет требуемые (оптимальные) свойства этих величин; расположение описания в программе в дальнейшем должно подчиняться требованиям выбранного языка программирования.

Для того, чтобы обозреть наши достижения, соберем все операторы, работающие с отражением, в программу:

```
программа ~ начало очистить(отражение);  
      DECLARE (X, Y) FIXED (2);  
      DO I=1 BY 1 TO 1000;  
        X=F(I); Y=G(I);  
        поставить знак (отражение, X, Y)  
      END;  
      напечатать (отражение)  
конец
```

В соответствии с правилами PL/1 описание вынесено из цикла.

2°. Чтобы дальше детализировать алгоритм, нам придется принять какие-то решения по структуре отражения. Собственно говоря, решение мы уже приняли раньше, когда договорились, что свойства отражения максимально соответствуют реальному изображению на бумаге, которое, как известно, состоит из 50 строчек, содержащих по 100 символов. Но торопиться не стоит, и, помня о минимальности шага детализации как алгоритма, так и данных, посмотрим, как конкретизируются наши операторы, если пока считать, что отражение состоит просто из 50 строчек.

Используя квадратные скобки для обозначения индексов и их граничных пар, принимаем:

отражение ~ строчки[0:49]

Тогда получим:

```
очистить ~ DO J=49 BY -1 TO 0;  
           чистка строчки (строчка [J])  
           END;  
поставить знак ~ знак в строчку (строчка[Y], X)  
напечатать ~ DO J=49 BY -1 TO 0;  
              печать строчки (строчка[J]);  
              переход к новой строчке  
           END;
```

Переход к новой строчке в нашем случае необходим в силу специфики операторов, управляющих печатью.

Можно было бы опять собрать полную программу, но повременим с этим,

3°. Для дальнейшей детализации принимаем:

строка ~ символы[0:99]

Тогда получаем:

чистка строки ~ DO K=0 BY 1 TO 99;

символ[K]:=пробел

END;

знак в строку ~ символ[X]:=звездочка

печать строки ~ DO K=0 BY 1 TO 99;

печать знака (символ[K])

END;

переход к новой строке ~ CALL MOVE;

печать знака ~ IF символ[K]=звездочка

THEN CALL NEXT(2);

ELSE CALL NEXT(1);

Перед тем, как сделать последний шаг, несколько укрупним операторы для обозримости. При этом следует обратить внимание на то, что операции над символами производятся в циклах, работающих с конкретными строками, координата (индекс) которых были определены на предыдущем шаге алгоритмизации. Обозначение конкретной строки, над символами которой производятся действия, будет присоединяться к обозначению символа-операнда с помощью точки (аналогично квалифицированному имени в PL/1). Например,

строка [J].символ[k]

обозначает k-й символ в j-й строке отражения,

Итак:

очистить ~ DO J=49 BY -1 TO 0;

DO K=0 BY 1 TO 99;

строка [J].символ [K]:=пробел

END;

END;

поставить звездочки ~ DO I=1 BY 1 TO 1000;

X=F(I); Y=G(I);

строка[Y].символ[X]:=звездочка

END;

напечатать ~ DO J=49 BY -1 TO 0;

DO K=0 BY 1 TO 99;

IF строка [J].символ[K]=звездочка

THEN CALL NEXT(2);

ELSE CALL NEXT(1);

END;

CALL MOVE;

END;

4°. Прежде, чем заняться представлением величин полученного алгоритма в терминах языка PL/1, следует задуматься о его эффективности. Оказывается, что с точки зрения минимизации печати, наш алгоритм не выдерживает никакой критики. Действительно, нет никакой необходимости выводить пробелы вплоть до 99-й позиции каждой строки, а следовало бы оканчивать печать на последней звездочке, присутствующей в строке. Кроме того, те строки, в которых нет ни одной звездочки, следовало бы просто пропускать.

Для того, чтобы можно было реализовать такую минимизацию печати, необходимо для каждой строки знать координату самой правой звездочки, поставленной в нее, т. е. максимальное значение X в каждой из 50 строчек; для строчек, в которых нет звездочек, этот максимум должен быть меньше минимального X , то есть меньше нуля.

Итак, каждая строка отражения помимо 50 символов должна содержать и величину *счетчик*, предназначенную для вычисления и хранения максимального X для данной строки. Таким образом, необходимо вернуться к п.3° и изменить некоторые блоки:

```

строка~{счетчик; символы [0:99]}
чистка строки~{счетчик:=—1;
                <дальше, как и ранее в п. 3°>}
знак в строку~{IF X>счетчик
                THEN счетчик:=X;
                <дальше, как и ранее>}
печать строки~{DO K=0 BY 1 TO счетчик;
                <дальше, как и ранее>}
```

В угловые скобки здесь заключены комментарии.

Как видим, при использовании обобщенных операторов значительно упрощается внесение изменений в алгоритм; например, в данном случае они затрагивают только те места алгоритма, где фигурирует термин «строка». Кроме того, запомним на будущее, что счетчик относится к конкретной обрабатываемой строке, которую нужно будет указать при дальнейшей конкретизации алгоритма.

5°. Для того, чтобы написать полученный алгоритм на PL/1, необходимо выразить используемые в нем величины и операторы через понятия PL/1. Отражение могло бы быть представлено и в виде двумерного массива, но более соответствует проведенной разработке такая структура в языке PL/1 (массив структур):

```

DECLARE 1 OTRAG, 2 STRING (0:49),
        3 COUNT FIXED (2),
        3 SYMBOL (0:99) BIT (1);
```

Звездочке поставим в соответствие значение '1'B, пробелу — '0'B.

Программа, оформленная в силу своих небольших размеров в виде блока, приобретает такой вид:

/*ГРАФИК ПАРАМЕТРИЧЕСКОЙ ФУНКЦИИ*/

```
BEGIN; DECLARE 1 OTRAG, 2 STRING (0:49),
                3 COUNT FIXED (2);,
                3 SYMBOL (0:99) BIT (1);
    DECLARE (X,Y) FIXED (2),
            (I,J,K) FIXED BINARY;
    CLEAR: DO J=49 BY -1 TO 0; /*ОЧИСТКА ОТРАЖЕНИЯ*/
        STRING (J).COUNT=-1;
        DO K=0 BY 1 TO 99:
            STRING(J).SYMBOL(K)='0'B;
        END;
    END;
    STAND: DO I=1 BY 1 TO 1000; /*ПОСТАВИТЬ ЗНАКИ*/
        X=F(I); Y=G(I);
        IF X > STRING (Y).COUNT
            THEN STRING(Y).COUNT=X;
        STRING (Y).SYMBOL(X)='1'B
    END;
    PRINT: DO J=49 BY -1 TO 0; /*ПЕЧАТАТЬ
                                ОТРАЖЕНИЕ*/
        DO K=0 BY 1 TO STRING(J).COUNT;
        IF STRING (J).SYMBOL (K)
            THEN CALL NEXT (2);
            ELSE CALL NEXT (1);
        END;
        CALL MOVE;
    END;
END /*БЛОКА*/;
```

З а м е ч а н и я . 1. Средства PL/1 позволяют упростить алгоритм очистки отражения. Если воспользоваться атрибутом начальных значений, то цикл очистки становится лишним, а COUNT и SYMBOL просто приобретают соответствующие атрибуты:

INITIAL ((50)—1) и INITIAL ((5000) '0'B)

2. Поскольку идентификаторы COUNT и SYMBOL используются в нашем блоке только в одном смысле, имя структурного массива STRING можно в квалифицированных именах везде опустить, если ваботаться не о мнемоничности имен, а только об их краткости. То

есть можно было бы выше задавать

COUNT(J), SYMBOL (J, K), SYMBOL (Y, X), COUNT (Y).

8. Э. Дейкстра предлагает, кроме того, и более экономичное решение, которое позволяет сократить и время, необходимое для очистки отражения. Действительно, при наличии счетчика очистка строчки осуществляется уже тем, что счетчику присваивается отрицательное значение. Нужно дополнительно позаботиться лишь о том, чтобы уже во время построения отражения при установке звездочки в какую-либо строчку отражения дополнительно устанавливать пробелы во все позиции, находящиеся левее, не содержащие звездочки и не очищенные ранее. Таким образом, заполнение отражения пробелами (см. цикл по K в чистке отражения или INITIAL для переменной SYMBOL) можно убрать, а оператор, вычисляющий максимальное значение X в строчке (см. IF-оператор в STAND), в нашем случае заменить на следующий:

```
DO WHILE (STRING(Y).COUNT < X);  
  STRING(Y).COUNT = STRING(Y).COUNT + 1;  
  STRING(Y).SYMBOL (STRING(Y).COUNT) = '0'B;  
END;
```

5°. Если алгоритм необходимо написать на алголе-60, то для представления отражения, очевидно, нужно использовать двумерный массив. Счетчик, для большей наглядности при записи алгоритма, удобно вынести из массива и образовать отдельный одномерный массив (типа целый); в противном случае двумерный массив должен иметь не логический (булевский) тип, а целый, что было бы более расточительным по памяти.

Конечно, если заранее знать о том, что придется писать программу на алголе, то на ранних стадиях алгоритмизации удобнее было бы в качестве вспомогательных использовать операторы алгола-60, а не PL/1. Но ввиду близости использованных операторов этих языков, программисты, владеющие обоими языками, все равно могли бы отдать предпочтение операторам PL/1 ввиду их большей универсальности.

Программа, оформленная в виде блока алгола-60, примет следующий вид:

```
begin comment ГРАФИК ПАРАМЕТРИЧЕСКОЙ  
  ФУНКЦИИ;  
  Boolean array OTRAG[0:49, 0:99];  
  Integer array COUNT[0:49];  
  integer x, y, i, j, k;
```

```

CLEAR: for j:=49 step -1 until 0
      do begin COUNT[j]:=-1;
              for k:=0 step 1 until 99
                do OTRAG[j, k]:=false
              end ОЧИСТКА ОТРАЖЕНИЯ;
STAND: for i:=1 step 1 until 1000
      do begin x:=f(i); y:=g(i);
              if x > COUNT[y]
                then COUNT[y]:=x;
                OTRAG(y, x):=true
              end УСТАНОВКИ ЗНАКОВ;
PRINT: for j=49 step -1 until 0
      do begin
              for k:=0 step 1 until COUNT[j]
                do if OTRAG[j, k] then NEXT (2)
                   else NEXT (1);

              MOVE
            end ПЕЧАТИ ОТРАЖЕНИЯ
end

```

5.2.2. Модульность и структурированность. Как видно из примеров, при нисходящем проектировании производится разбиение программы или ее частей на функционально независимые фрагменты и программирование их отдельно друг от друга. Тем самым достигается независимость фрагментов программы и в крупном — для случая, когда они реализуются блоками (простыми или процедурными), и в мелком — при реализации их несколькими операторами конкретного алгоритмического языка.

Использование на каждом уровне детализации только стандартных структурированных схем обеспечивает тем самым получение структурированной программы, содержащей только вложенные друг в друга и стандартно объединенные фрагменты модуля. При этом главной является не техническая сторона структурного программирования (запись алгоритма с использованием только стандартных структур), а применение крупных алгоритмических операторов и данных, разбиваемых в ходе дальнейшей разработки на все более мелкие составные части. То есть можно сказать, что речь идет, по существу, о необходимости освоения специального «структурированно-нисходящего» мышления для проектирования программ. Для преодоления трудностей, возникающих в ходе такого проектирования, как раз и служит детализация с помощью мелких шагов.

Таким образом, нисходящий способ разработки программ позволяет получать программы, обладающие свойствами модульности и структурированности.

После реализации независимых частей программы на достаточно детальном уровне, особенно на конкретном алгоритмическом языке, может проводиться и обратный процесс — объединение разработанных фрагментов в блоки или модули для более наглядного представления реализуемого алгоритма и для уяснения информационных связей в проектируемой части программы (см. второй пример или передачу SYM в первом примере).

5.2.3. Блок-схема и псевдокод. Обратим внимание на то, что в примерах совсем не использовались блок-схемы, которые раньше рекомендовались для разработки алгоритмов. Можно предложить следующее объяснение такому факту.

Раньше утверждалось, что наглядное представление логической структуры программ с помощью блок-схем особенно полезно или даже необходимо для алгоритмов, имеющих сложное строение. Поскольку применение нисходящего проектирования позволяет максимально упростить структуру программ, сводя ее к последовательности стандартных схем, острая необходимость в блок-схемах пропадает.

Блок-схемы могут быть полезны на самой начальной стадии разработки алгоритма программы (или ее отдельных блоков) для первых, обычно карандашных, набросков решения и систематизации своего мышления во время исследования поставленной задачи и уяснения проблем, возникающих при ее алгоритмизации. Блок-схемы были необходимы в то время, когда программирование производилось в машинных кодах или на языке ассемблера без использования макро-средств и идей структурированного программирования. При документировании программ блок-схемы можно использовать для формулировки основного подхода к решению задачи или дополнительного пояснения к алгоритму; описываемому формально на алгоритмическом языке.

При нисходящей разработке алгоритма обычно вместо блок-схем используются конкретные или условные алгоритмические языки высокого уровня. Конкретный алгоритмический язык (вначале, может быть, и отдельные его средства) применяется на поздней стадии разработки алгоритма, причем применяется обычно тот язык, который выбран для реализации программы. Но если этот язык является мало пригодным для нисходящего проектирования, то могут быть использованы средства и другого языка, с последующим их переводом в конструкции языка, выбранного для программирования (конечно, такой перевод должен быть достаточно механистическим).

На более ранних стадиях используется условный алгоритмический язык; или как его называют — *псевдокод*, который отличается от конкретных алгоритмических языков следующими особенностями.

Синтаксис псевдокода может быть гораздо более свободным, чем у алгоритмических языков, поскольку главным его назначением является выражение общих черт разрабатываемого алгоритма, подвергаемых в дальнейшем уточнению. На этой стадии формулировки алгоритма не важны синтаксические детали конструкций, которые в реальных алгоритмических языках играют существенную роль. Разработчик выбирает такие понятия и конструкции для псевдокода, которые по его мнению помогут выразить алгоритм решения в максимально простой форме, отвечающей специфике решаемой задачи. Желательно, конечно, чтобы конструкции псевдокода были похожи на типовые конструкции реальных алгоритмических языков, чтобы алгоритмы, выраженные на псевдокоде, могли без пояснений понимать многие программисты в ходе дальнейшей разработки или модернизации.

Псевдокод по своему характеру является промежуточным между естественным (в нашем случае русским) языком и алгоритмическим языком. Чем дальше продвигается разработка, тем больше в используемом псевдокоде появляется черт алгоритмического языка, как в отношении его операторов, так и обрабатываемых данных.

5.3. Особенности применения

5.3.1. Трудности и преимущества. Одной из основных причин, препятствующих использованию нисходящего проектирования программистами, является их привычка применять в своей работе только операторы или команды известных им языков программирования и мелкие, принятые в этих языках, типы данных. Использование обобщенных данных и укрупненных операторов требует определенного навыка в применении элементов абстрактного мышления. Но именно обобщенные операторы и данные позволяют в максимально простом виде выразить структуру алгоритма программы, найти простоту в сложных и запутанных взаимодействиях величин решаемой задачи.

Выше уже говорилось о том, что программисту необходимо уметь предвидеть последствия принимаемых им решений, а это является особенно трудным на начальных этапах алгоритмизации и на первых порах использования нисходящего проектирования. Вообще, можно отметить, что нисходящее проектирование требует более напряженной умственной деятельности на ранней стадии разработки, чем при обычном «восходящем» способе разработки, когда программист сначала основное внимание уделяет работе над отдельными локальными участками программы, мало заботясь об оптимальности общей структуры программы или блока. Усилия, затрачиваемые програм

мистом в ходе разработки программы, можно выразить в виде графика, приведенного на рис. 12.

Нисходящее проектирование предполагает выявление в самом начале разработки основных, главных функций создаваемой программы и их реализацию до установления других, вспомогательных

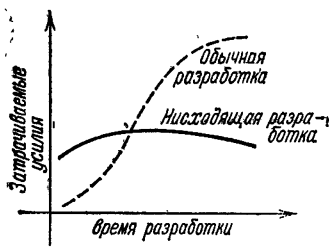


Рис. 12. Затрачиваемые усилия на разработку.

функций. Выявление главных функций программы среди множества других, содержащихся в формулировке задачи, и почему-либо бросающихся в глаза, также является не простой проблемой. По существу, единственным материалом для такого выявления является формулировка задачи, и поэтому для реальных задач техническое задание на разработку должно облегчать понимание основных функций задачи и реализацию первых этапов нисходящего проектирования. Имеет большой смысл, чтобы техническое задание само было построено по нисходящему принципу, с изложением сначала основных функций, а затем — остальных.

Заметим также, что если в описании готовой программы, изготовляемой в самом конце разработки, отразить и процесс ее разработки с указанием основных этапов детализации алгоритма и данных, то это значительно облегчает изучение программы в ходе ее дальнейшего использования и модернизации. Удобно, в частности, чтобы комментарии в тексте программы формулировались в терминах обобщенных понятий, используемых при разработке алгоритма на различных этапах.

Здесь речь шла об этапе алгоритмизации, но использование нисходящего проектирования позволяет значительно легче и быстрее проводить тестирование и отладку программы. Ввиду логической простоты структуры программы, подавляющее большинство логических ошибок программы оказывается возможным находить за столом без использования машины. Машина, следовательно, будет в основном использоваться только для доказательства правильности работы программы (на этапе контроля), а не для локализации ошибок, что, как правило, не достижимо при обычном способе программирования.

Появление нисходящего проектирования знаменует такой этап развития вычислительной техники и программирования, когда обращают внимание не только на эффективность работы программы, но и на такие ее характеристики, как надежность и простота эксплуатации. Эффективность составления программ, возможность получать работоспособные и надежные программы в сжатые сроки, является не менее важной стороной проектирования программ, чем эффективность их работы. То есть можно сказать, что в большинстве случаев основной заботой разработчика наряду с эффективностью программы должна быть и эффективность программирования. Именно оптимальное сочетание усилий, затрачиваемых на проектирование и на обеспечение эффективности программы, характеризует искусство программиста. Во всяком случае, ушло в прошлое то время, когда квалификация программиста оценивалась многими по количеству остроумных трюков и степени запутанности его программы, когда понятие программы было синонимом сложности и непонятности для всех, кроме ее автора (а часто и для него самого).

Нисходящее проектирование, облегчая в значительной степени отладку — наиболее трудный этап разработки программ, — позволяет рядовым и начинающим разработчикам сравняться с более сильными и опытными по эффективности отладки программ и сложности решаемых задач, а опытным — повысить надежность сложных программ, то есть уменьшить количество необнаруженных ошибок. Содействие нисходящего проектирования эффективному поиску ошибок в программе можно объяснить и тем, что зафиксированные на бумаге различные по детализации уровни алгоритма помогают разработчикам быстро переключать свое внимание от общей структуры программы на другие гораздо более конкретные ее представления. А именно такая возможность (или способность программиста) является, в конечном счете, главным условием успешной локализации ошибок и, в целом, разработки программ.

5.3.2. Нисходящее тестирование. Под нисходящим проектированием понимается обычно общая методика разработки программ, касающаяся как этапа собственно проектирования (см. п.0.1), так и алгоритмизации, программирования и контроля программ (тестирования). Причем, вместо термина «программирование» используется «кодирование», для подчеркивания простого, механического характера работ на этом этапе. То есть нисходящий принцип может быть использован не только на стадии начального проектирования и алгоритмизации, но и при кодировании и тестировании.

Под *нисходящим кодированием* понимается такая организация работ, при которой кодирование, по возможности, производится в коде или сразу после получения какого-либо очередного уровня алгоритма, как например во втором примере (см. 5.2.1). Обычное вос-

ходящее кодирование производится после детализации алгоритма вплоть до самого последнего уровня (см. первый пример).

Использование элементов конкретного алгоритмического языка высокого уровня на ранних стадиях разработки позволяет формулировать детализируемый алгоритм четко и точно, что предупреждает возникновение случайных ошибок при проектировании.

Кроме того, нисходящее кодирование позволяет осуществить *нисходящее тестирование*, под которым понимается тестирование, выполняемое сразу после реализации каждого шага алгоритмизации, до проведения следующего шага детализации. Нисходящее тестирование противопоставляется обычному, «восходящему» тестированию, при котором тестирование данного блока начинается только после проведения последнего уровня его детализации, а тестирование всей программы — после контроля всех блоков программы.

Поскольку при нисходящем тестировании детали функции многих блоков обычно еще не определены и эти блоки не могут быть пока закодированы, такое тестирование возможно только при замене не конкретизированных еще блоков или их частей на имитаторы, в задачу которых входит поддерживать функционирование всей проверяемой программы. Имитаторы принимают передаваемые им данные и передают их после контрольной печати и некоторой переработки (возможно и пустой) следующему или вызываемому блоку. На первых этапах алгоритмизации тестирование сводится к самой общей проверке взаимодействия основных блоков программы, к проверке соответствия форматов передаваемых и принимаемых данных во взаимодействующих блоках.

Нисходящее тестирование позволяет выявить ошибки в реализации основных функций программы уже на ранних стадиях работы над программой, до начала реализации других, второстепенных функций. Таким образом, открывается возможность начинать активную отладку программ на машине и находить самые принципиальные ошибки гораздо раньше, чем обычно. Даже для достаточно сложных программ можно получить первые тестовые результаты в самый первый день разработки алгоритма. Например, уже на первой стадии проектирования имеет смысл проверить установленные в программе самые внешние характеристики основных наборов данных, служащих для ввода-вывода или являющихся интерфейсом между основными частями программы. В ОС ЕС при этом удобно использовать фиктивные наборы данных (с именем `NULLFILE` или с параметром `DUMMY` в директиве `DD`).

Каждый новый шаг детализации алгоритма усложняет и конкретизирует характер отладки, что позволяет постепенно усложнять контроль и отладку программы, придавая этим этапам более регулярный характер и, тем самым, значительно облегчая их прове-

дение. Конечно, подготовительная работа для проведения тестирования на каждом шаге, а также составление и проверка самих имитаторов («затычек», «заглушек») требует дополнительных усилий, и поэтому к нисходящему тестированию приступают не на самых первых шагах разработки, а несколько позже, когда логическая структура уже достаточно детализирована. Тем самым, для несложных программ нисходящее тестирование может оказаться невыгодным из-за сравнительно большой подготовительной работы. Практически, по различным причинам обычно осуществляется смешанная стратегия отладки, когда часть модулей отлаживается независимо от центральной части, а затем подключается к ней. Такими причинами могут быть следующие: наличие уже готовых модулей; отсутствие достаточного количества машинного времени, для того чтобы проверка главного модуля обгоняла проверку других модулей; наличие модулей, реализующих критические функции, от успеха разработки которых зависит успех всей программы и т. п.

Использование имитаторов при контроле программ обсуждалось и раньше (см. 1.2), но там оно не носило систематического характера и являлось лишь элементом смешанной стратегии контроля. Там же можно найти и другие сведения, касающиеся особенностей и преимуществ использования имитаторов при тестировании.

ГЛАВА 6

ПРИМЕРЫ РАЗРАБОТКИ И ОТЛАДКИ

6.1. Преобразование текста

6.1.1. Постановка задачи. Предположим, что техническое задание на разработку программы выдано в виде формулировки задачи, приведенной в 5.1.2. Конечно, эта формулировка не вполне соответствует требованиям приведенным в разделе 0.1 (см. п. 0°), но условно отнесем это несоответствие за счет неопытности заказчика (или проектировщика, выдавшего задание). Все недоработки в ТЗ придется компенсировать в начале предстоящей разработки программы, то есть в ходе составления проекта (см. ниже).

6.1.2. Проект. Программа TRANSF, выполняющая преобразование текста согласно ТЗ (см. 6.1.1 и 5.1.2), составляется на фортране (или на PL/1) для ЕС ЭВМ и оформляется в виде подпрограммы (внешней процедуры). Входной текст состоит из символов ЕС ЭВМ. Ограничений на длину текста программа не накладывает, но следует иметь в виду ограничения, накладываемые подпрограммой OUTSYM, используемой в подпрограмме TRANSF.

Если текст начинается с пробела (или точки) или если количество символов во вводимом слове превышает 20, а также если при обработке обнаруживается, что в конце вводимого текста отсутствует точка, то выводится диагностика, преобразование прекращается и происходит возврат в вызывающую программу. При этом передается и величина, характеризующая тип ошибки или ее отсутствие (в последнем случае величина равна нулю); передача осуществляется с помощью параметра-аргумента процедуры TRANSF. Вывод диагностики производится путем обращения к подпрограмме DIAGN, которой в качестве аргумента задается номер выводимой диагностики.

Общая схема взаимосвязи подпрограмм-модулей приведена на рис. 13.

Характеристика подпрограмм. Подпрограмма TRANSF (ERRTYP). Преобразование текста, получаемого от INSYM и выдаваемого через OUTSYM; ERRTYP — выходная величина, характеризующая тип возникшей ошибки (величина равна нулю, если окончание подпрограммы нормальное).

Подпрограмма DIAGN (NUMERR). Вывод диагностики по заданному номеру NUMERR.

Подпрограмма INSYM. Функция, имеющая значение очередного символа преобразуемого текста.

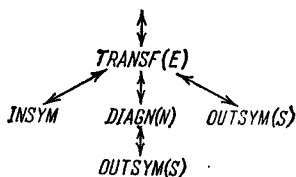


Рис. 13. Общая схема взаимодействия подпрограмм.

Подпрограмма OUTSYM (SYM). Вывод SYM.

Проект инструкции. Подпрограмма TRANSF служит для осуществления специфического преобразования текста, состоящего из разделенных пробелами слов длиной не более, чем 20 символов, и кончающегося точкой; текст не может начинаться с пробела или точки. Преобразование заключается в том, что количество пробелов между словами текста сводится к одному, а символы четных слов текста располагаются в обратном порядке; пробелы перед точкой удаляются. Например, текст

СОКРАТИТЬ_П_ПРОБЕЛЫ_И_П_

П_ПЕРЕВЕРНУТЬ_П_ЧЕТНЫЕ_СЛОВА_П_.

преобразуется в следующий:

СОКРАТИТЬ_Ы_ЛЕБОРП_И_ЬТУНРЕВЕРЕП_ЧЕТНЫЕ
П_АВОЛС.

Подпрограмма TRANSF получает исходный текст, обращаясь к подпрограмме INSYM, и выводит преобразованный текст с помощью подпрограммы OUTSYM, и поэтому перед работой с подпрограммой TRANSF необходимо ознакомиться с инструкциями для упомянутых подпрограмм.

Если преобразуемый текст не удовлетворяет перечисленным ранее требованиям, то выдается диагностика и преобразование прекращается; целой переменной ERRTYP, являющейся аргументом, при выполнении TRANSF присваивается одно из следующих значений:

ERRTYP = 0: нормальное окончание преобразования;

ERRTYP = 1: текст начинается с пробела или точки;

ERRTYP = 2: в слове более 20 символов.

Если в конце текста отсутствует точка, то эта ошибка должна обнаруживаться в подпрограмме INSYM при исчерпании входного потока.

Диагностика выдается с помощью подпрограммы OUTSYM. На объем преобразуемого текста подпрограмма TRANSF ограничена не накладывает.

План тестирования. Система тестов должна проверять работу программы для перечисленных ниже вариантов исходного текста. Текст должен состоять и из нескольких слов, и из одного слова. Слова должны содержать и несколько символов, и 1 символ, и 20 символов, причем каждое из таких слов должно стоять как на четном, так и на нечетном местах. Перед точкой должно находиться и несколько пробелов, и ни одного. Следует проверить случаи, когда программа должна выдавать диагностику (см. выше).

При отладке вместо INSYM и OUTSYM используются простейшие имитаторы их работы; преобразованный текст должен выдаваться имитатором OUTSYM на печать.

6.1.3. Алгоритм. Алгоритм и ход его разработки приведен в 5.1.2 (см. п. 1°+5°). После каждого шага детализации алгоритма производится его проверка (или просмотр) за столом для выявления ошибок, внесенных на очередном шаге алгоритмизации. Поскольку ввиду достаточной простоты алгоритма оказалось возможным при программировании объединить все фрагменты (модули) алгоритма в единый блок, прокрутку удобнее будет произвести уже по готовой программе (см. ниже 6.1.4).

В необходимых местах алгоритма должна быть вставлена наменная ранее в 6.1.2 защита от ошибок в исходном тексте. После ввода первого символа нужно проверить его на пробел и точку:

если символ пробел или точка то ошибка (1)

В ходе ввода слова нужно проверять, не превысило ли двадцати количество введенных символов. Это можно сделать, изменив, например, следующим образом заголовок цикла:

*для $i=1$ шаг 1 до 20 пока (символ не пробел и не точка)
цикл*

После окончания ввода слова нужно проверить на пробел или точку последний введенный символ:

если символ не пробел и не точка то ошибка (2)

Фиксация ошибки включает в себя следующие действия:

а) присваивание параметру ERRTP возвращаемого значения (1 или 2);

б) обращение к подпрограмме DIAGN для печати нужной диагностики;

в) возврат в вызывающую программу.

Если выполнять эти действия непосредственно в соответствующих местах программы, то будет нарушено правило структурированности, требующее, чтобы из модуля был только один выход. Использование оператора GO TO (для перехода на единственный оператор возврата) также нарушило бы структурированность программы. Остается только организовать защиту от ошибок в нашей подпрограмме, например, следующим образом (см. 3° и 4° в п. 5.1.2):

программа~

```

~начало увертюра; ERRTYP:=0; ввод символа;
  если символ не пробел и не точка то
    пока (не точка и ERRTYP = 0) цикл
      начало запоминание 1-го символа слова;
      ввод слова с запоминанием;
      если символ пробел или точка то
        начало пропускание пробелов;
        вывод порции;
      конец
    иначе ERRTYP:=2
  конец
  иначе ERRTYP:=1;
  если ERRTYP ≠ 0 то вызов DIAGN(ERRTYP);
  возврат
конец

```

Мы добились структурированности нашего модуля, но приходится отметить, что его внешняя структура заметно усложнилась от использования двух условных операторов, включающих в себя основные части модуля. Поэтому в нашем случае, пожалуй, будет оправданным допустить несколько выходов из модуля TRANSF, нарушив тем самым одно из правил структурированности (см. 4.3.4 п. 8)).

Таким образом:

ошибка (n) ~ {ERRTYP:=n; вызов DIAGN(n); возврат}

Можно было бы присваивание переменной ERRTYP производить и в подпрограмме DIAGN, если задавать ERRTYP в качестве второго параметра подпрограммы DIAGN. Кроме того, можно было бы также обращаться к DIAGN и для ERRTYP = 0, печатая, например, НОРМАЛЬНОЕ ЗАВЕРШЕНИЕ TRANSF.

План отладки. Для полученного алгоритма теперь следует выбрать необходимые отладочные средства и наметить такой способ их использования, чтобы можно было по получаемым отладочным данным легко локализовать ошибки в отлаживаемой программе

В первую очередь, очевидно, необходима печать введенного слова в узле, расположенном между вводом и выводом порции, чтобы можно было отделить ошибки реализации ввода от ошибок вывода порции. Помимо введенного слова нужно печатать также и другие данные, которые были получены в процессе ввода и используются затем при преобразовании и выводе, например, количество символов в слове. Кроме того, имеет смысл дополнительно печатать последний символ порции, присоединяемый к следующему слову, ввиду определенной логической трудности в алгоритме, связанной с обработкой этого символа.

Ввиду того, что преобразованный текст будет с помощью имитатора OUTSYM выдаваться на печать, ошибки преобразования, по видимому, можно будет легко найти по отпечатанным результатам, и поэтому отладочной печати для локализации ошибок вывода порции не требуется. Правильность пропуска пробелов при вводе также легко проверяется по выведенному тексту.

Проверку правильности стыковки с процедурой INSYM можно осуществить, печатая получаемые от этой процедуры символы. Причем, так как каждый вводимый символ в составляемой подпрограмме должен сохраняться до следующего цикла в виде значения некоторой переменной, можно для печати этого значения применить арифметическое слежение (см. 2.1.3, А). Проверку правильности передачи символа в процедуру OUTSYM следовало бы предусмотреть в самой процедуре; в противном случае придется изыскивать способы печати из TRANSF передаваемых символов.

Перед переходом к программированию следует определить алгоритмы имитаторов и уточнить общую схему взаимодействия подпрограммы TRANSF с имитаторами INSYM, OUTSYM и DIAGN. Можно предложить следующие достаточно простые алгоритмы имитаторов для INSYM и OUTSYM.

процедура INSYM;

общий (цел m , симв $a[1:80]$);

$m := m + 1$;

если $m > 80$ то {печать('ENDFILE'); стоп};

возврат (a_m);

конец

процедура OUTSYM(s);

общий (цел k , симв $b[1:80]$);

$k := k + 1$;

$b_k := s$;

возврат;

конец

Имитатор INSYM при каждом обращении к нему, в качестве очередного символа исходного текста выдает следующий элемент

массива a . Имитатор OUTSYM очередной выводимый символ за-
сылает в следующий по порядку элемент массива b . Величины a ,
 b , m , k являются «общими» (COMMON в фортране, EXTERNAL
в PL/1): они известны в нескольких подпрограммах и сохраняют
свои значения после выхода из подпрограмм INSYM и OUTSYM.
Предполагается, что предварительное заполнение массива a и печать
массива b , а также обнуление вели-
чин m и k будут производиться
в головном имитаторе, из которого
производится вызов основного
модуля TRANSF. Такая организа-
ция дает возможность за один
выход на машину протестировать
программу сразу для нескольких
исходных данных, которые могут
быть введены по очереди с перфо-
карт в головном модуле-имитато-
ре. Длина тестового исходного текста ограничивается ради простоты
имитаторов 80 символами, что допустимо для той системы тестов,
которая была запланирована раньше.

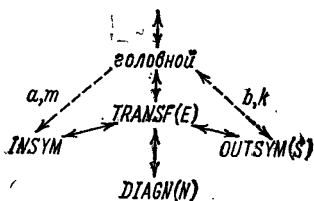


Рис. 14. Взаимосвязь модулей.

Алгоритм головного модуля:

начало

общий (цел (m, k) , симв $(a[1:80]; b[1:80])$);

пока ввод не исчерпан цикл

начало ввод(a);

$m, k := 0$;

вызов TRANSF(e);

вывод (b);

конец

конец

На рис. 14 показана взаимосвязь модулей (или их имитаторов),
участвующих в отладке.

В качестве имитатора для DIAGN можно взять подпрограмму,
которая просто печатает аргумент—номер выводимой диагностики:

процедура DIAGN(N);

целый N ;

печать ('ОШИБКА', N);

конец

6.1.4. Программа. Программа была приведена в 5.1.2. п. 6^{оо}
(или 6^о для PL/1), но для обнаружения ошибок во входном тексте,
нужно добавить некоторые операторы (см. 6.1.3). Например, после
ввода первого символа нужно вставить:

```

IF(SYM .NE. BLANK .AND. SYM .NE. POINT) GO TO 05
  ERRTP=1
  CALL DIAGN(1)
  RETURN

```

05

или для PL/1:

```

IF(SYM=' ' ! SYM='.') THEN
  DO; ERRTP=1; CALL DIAGN(1); RETURN; END;

```

Подобные же операторы вставляются перед пропуском пробелов (см. 6.1.3). В начало программы добавляется ERRTP = 0.

Ниже приводятся тексты программ для некоторых имитаторов.
Для фортрана.

```

C /* ГОЛОВНАЯ ПРОГРАММА-ИМИТАТОР */
COMMON /IN/A,M /OUT/B,K
INTEGER *2 A(80), B(80), ERRTP/99/, ZABOI/'#' /
INTEGER M, K, MK, INF/1/, INI/2/, TE/0/
00010 DO 80 MK=INF, INI, TE
      READ (5, 20, END=90) A
20    FORMAT (80 A1)
C /**** ПОЗДНЯЯ ВСТАВКА—СМ. П. 7.1.5 ****/
      WRITE (6, 70) A
      DO 60 I1=1, 80, 1
        B(I1)=ZABOI
60    CONTINUE
C /***** КОНЕЦ ВСТАВКИ *****/
      M=0
      K=0
      CALL TRANF(ERRTP)
      WRITE(6,70) ERRTP, B
70    FORMAT (1X, I5, 80, A1)
80    CONTINUE
90    STOP
      END

```

Для присваивания общим величинам A, B, M, K начальных значений нужно воспользоваться подпрограммой BLOCK DATA. Выход из цикла производится по оператору READ в случае исчерпания входного тестового потока.

Для PL/1.

```

/* ГОЛОВНАЯ ПРОГРАММА-ИМИТАТОР */
HEAD: PROCEDURE OPTIONS (MAIN);
  DECLARE(M,K) EXTERNAL INITIAL (9999),
    (A(80), B(80)) CHAR(1) EXT INIT((80) (1)'#'),

```

```

ERRTYP FIXED INITIAL (9999),
TRANSF ENTRY (FIXED);
ON ENDFILE(SYSIN)
  BEGIN; PUT SKIP LIST ('КОНЕЦ ТЕКСТА');
                                STOP; END;
CYCL: DO WHILE (2 > 1);/*ВЫХОД ПО ENDFILE*/
  GET EDIT(A) (80 A(1));
  /**** ПОЗДНЯЯ ВСТАВКА—СМ. П. 7.1.5 ****/
  PUT SKIP EDIT(A) (80 A(1));
  B = '#';
  /**/
  M=0; K=0;
  CALL TRANSF(ERRTYP);
  PUT SKIP EDIT(B) (80 A(1));
  PUT DATA (ERRTYP);
END CYCL;
END HEAD;

```

Выход из цикла производится по оператору ON.

Остальные подпрограммы для имитаторов читатель может составить в качестве упражнения.

Средства отладки. В составленную программу TRANSF теперь нужно в соответствии с наметками, приведенными в п. 6.1.3, вставить отладочные операторы. Отладочный пакет для подпрограммы TRANSF на фортране будет выглядеть следующим образом:

```

% IF  $\square 1 = 1$  % THEN % DO;
  DEBUG
    AT 08
    WRITE (6, 909) SYM
909    FORMAT(1X, 'УЗЕЛ1:', A1)
    AT 40
    WRITE(6, 940) WORD, SYM, L
940    FORMAT(1X, 'УЗЕЛ2:', 20A1, 5X, A1, I5)
% END/* IF */;

```

Арифметическое слежение за SYM (режим INIT (SYM)) в данном случае не применимо, поскольку печать SYM происходила бы в виде числовых значений в соответствии с описанием SYM (INTEGER*2). Метка 08 должна быть приписана к оператору $L = L + 1$ в программе; метка 10 не может быть использована, так как она принадлежит невыполняемому оператору.

Для PL/1 после описаний нужно вставить следующие операторы:

```

% IF  $\square 1 = 1$  % THEN % DO;
ON CHECK(OUT) PUT SKIP EDIT

```

('УЗЕЛ1:', WORD, SYM, 1)
(A, 20 A(1), X(5), A(1), F(5));

% END /*IF CHECK */;

Кроме того перед заголовком процедуры TRANSF добавляется:
% IF $\square 1 = 1$ % THEN % DO;
(CHECK(OUT));
(CHECK(SYM));
% END /*IF CHECK */;

Чтобы следить за символами, передаваемыми в модуль OUTSYM, можно воспользоваться слежением за обращением к этому модулю, то есть добавить префикс (CHECK (OUTSYM)); и оператор

ON CHECK (OUTSYM) PUT EDIT (BUKWA (K), ':') (2 A(1));

Для включения и отключения отладочной печати и в фортране и в PL/1 используются препроцессорные средства. Описание препроцессорной переменной $\square 1$ и присваивание нужного значения удобно осуществить перед началом текста подпрограммы TRANSF. Когда подпрограмма будет помещена во внешнюю память, то в качестве транслируемого (препроцессорного) текста в шаге задания достаточно будет оставить только три следующих оператора (см. П. 6):

% DECLARE $\square 1$ FIXED;
% $\square 1 = 1$ /* ИЛИ $\square 1 = 0$ */;
% INCLUDE PTRANS /* ИМЯ РАЗДЕЛА */;

Печатью можно было бы управлять и по условию, но в нашем случае, видимо, пришлось бы отказаться от того, чтобы перед задаваемым для преобразования текстом находилось логическое (или арифметическое) значение, управляющее печатью; его нужно задавать в операторе программы. Для отключения одной из двух отладочных печатей можно воспользоваться утилитой UPDATE или снова прибегнуть к препроцессорным средствам. Слежение, видимо, понадобится на первой стадии автономной и, может быть, при стыковочной отладке. Напомним, что поскольку печать при слежении происходит на ЕС для каждого значения через строчку, то использование такого средства для больших текстов было бы нерационально. Но если в PL/1 использовать оператор, например, вида

ON CHECK (SYM) PUT EDIT (SYM, ':') (2 A (1));

то можно в одной строчке напечатать несколько результатов слежения.

Отладочную печать для проверки правильности работы самих имитаторов (и головной подпрограммы), пожалуй, можно не предусматривать ввиду их простоты; в случае необходимости ее можно

будет вставить дополнительно. Однако, в соответствии с общими правилами, следует в головной подпрограмме печатать значение массива А после ввода (см. выше «позднюю» вставку).

Тесты. В соответствии с планами, изложенными в п. 6.1.2, можно предложить следующую систему тестов (для краткости приведены только исходные тексты):

- 1) Я□ПРОВЕРЯЮСОСТАВЛЕННУЮ□□ПРОГРАММУ□□□.
- 2) ПР—МУ□□Я□□ПРОВЕРЯЮСОСТАВЛЕННУЮ.
- 3) Я.
- 4) □ОШИБКА.
- 5) .123.
- 6) 1□+12345678901234567890.
- 7) POINT

Тесты 1 и 2 можно считать типичными; в них содержатся слова трех видов: с минимальной, максимальной и средней длиной, причем слова стоят и на четном, и на нечетном местах. Текст 3 задает вырожденный случай. Тесты 4÷7 проверяют реакцию программы на ошибки в исходном тексте.

Так как и для правильного текста и для ошибочного выполнение каждого теста (кроме седьмого) будет кончаться возвращением в головной модуль, то за один пропуск программы можно надеяться проверить сразу все тесты.

Прокрутка. Для минимизации работы при ручной прокрутке программы возьмем следующий текст:

Я□ОНА□□ТЫ□□□ОН□.

Имеет смысл прокрутить отлаживаемую подпрограмму вместе с имитаторами, так как и в них могут быть ошибки. Процесс прокрутки для фортрانا отражен на рис. 15. Значком ∞ обозначена произвольная информация; три точки указывают на повторение предыдущего знака, знак \ указывает на стирание предыдущего значения; квадратные скобки ([и]) и вертикальная черта | отделяют элементы массива.

В качестве второго, дополнительного текста для прокрутки взят тест № 5 из списка тестов (см. выше).

Результаты прокрутки показывают, что программа, повидимому, правильна, но печать полученных тестовых результатов будет производиться по WRITE с «мусором» в конце текста; для удобства разбора тестовых результатов имеет смысл добавить в головную программу очистку массива В. Тест, использованный для прокрутки, имеет смысл присоединить к разработанной ранее системе тестов, в качестве самого первого теста.

Для PL/1 содержание прокрутки будет иметь тот же характер, что и на рис. 15, исключая следующие моменты:

Можно догадаться, что первая буква каждого слова куда-то пропадает, а в конце выводимых слов появляются два лишних символа. По выданным промежуточным результатам узнаём, что первые элементы массива BUKWA в узле OUT равны \square , P и P, в то время как слежение за SYM показывает, что от INSYM были получены правильные символы: Я, П и П. Значит, потеря первых символов слов происходит не в цикле вывода и не в модуле INSYM, а в цикле с меткой IN. После внимательного просмотра цикла можно, например, обнаружить, что 2 оператора, составляющих тело этого цикла, оказались почему-то переставленными. Что же касается лишних выводимых символов в конце каждого слова, то обратившись к печатаемым значениям I в узле OUT, обнаружим, что они равны 2, 21, 10 вместо правильных 1, 20, 9. То есть, ошибка находится не в модуле OUTSYM и не в цикле вывода (OUT), а опять же в цикле ввода. Ошибка, очевидно, состоит в том, что при программировании мы не учли следующую особенность цикла в PL/1: значение параметра цикла продвигается до проверки, осуществляемой по WHILE. Для исправления этой ошибки необходимо уменьшить на единицу значение I после выхода из цикла IN, добавив после него оператор $I = I - 1$. Найденные ошибки вполне объясняют полученные результаты, и можно перейти к анализу следующих тестовых результатов.

Без промежуточных результатов даже в такой небольшой программе быстро найти имеющиеся ошибки было бы значительно труднее. Если бы ошибок оказалось не две, а больше, или программа была бы не структурированной, то трудность поиска ошибок стала бы еще выше.

Фортран. Предположим, что программа на фортране после про пуска на машине напечатала следующие результаты (с учетом нулевого теста):

```
Я□ОНА□□ТЫ□□□ОН□,
ENDFILE
```

Так как печать в узлах отсутствует, то следовательно, конец исходных данных обнаруживается при вводе 1-го или 2-го символов текста; это заключение справедливо, если есть уверенность в том, что обращение к отладочным средствам произведено правильно. Обратимся к модулю INSYM, в котором производится печать диагностики ENDFILE. Эта диагностика печатается при $m > 80$, что в нашем случае может произойти только тогда, когда m имеет почему-либо неправильное значение. Для проверки этой гипотезы нам не хватает печати значений величины m (M), которая передается из модуля в модуль с помощью аппарата общих величин (COMMON). Понадеявшись на то, что алгоритмы имитаторов очень просты, мы

не предусмотрели в них никакой отладочной печати. Тем самым было нарушено правило, по которому требуется, чтобы в каждом модуле производилась отладочная печать всех получаемых из других модулей данных, к которым нужно отнести не только аргументы (параметры), но и общие данные.

Вставив печать величин m и a (в модуль INSYM), а также k и s (в модуль OUTSYM), можно легко обнаружить, что m передается из головного имитатора неверно. Это может являться, например, следствием ошибки в операторах присваивания $m:=0$ или $m:=m+1$, но это мало вероятно. Кроме этого, m встречается только в описании. Обратив свое внимание на описание общих величин в головном имитаторе и в INSYM, можно обнаружить, что порядок описания величин a и m (так же как b и k) в этих модулях не совпадает. Такая перестановка, которая не допустима в фортране, и вызывает ошибку.

6.2. Перевод программы

Ниже приводится несколько первых шагов нисходящего проектирования вполне реальной и достаточно сложной задачи.

6.2.1. Техническое задание. Разработать программу для перевода программ, написанных на алголе TA-1M [16], в алгол ЕС [15].

Исходная программа находится на перфокартах в кодировке, соответствующей внешним устройствам для ЭВМ типа М-220. Полученная программа выдается на печать и на выбранный внешний носитель ЕС ЭВМ (магнитные диск или лента, перфокарты).

З а м е ч а н и е. Следует учесть, что в будущем возможен ввод исходного текста с магнитных лент и дисков в кодировке, которая в настоящее время еще не определена.

Самая общая схема работы программы приведена на рис. 16.

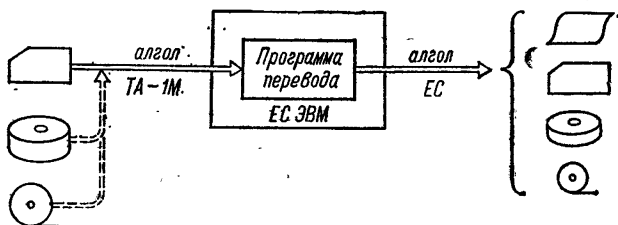


Рис. 16. Общая схема работы программы перевода.

6.2.2. Проект. Перед проектированием программы проводится анализ задания.

Анализ задания. Основная трудность реализации программы состоит в переводе обращений к стандартным подпрограммам-кодам,

используемым, в частности, и для ввода-вывода в ТА-1М (идентификаторы таких подпрограмм имеют вид P0042, P1041 и т. п.); большинство таких подпрограмм в СМО ЕС ЭВМ отсутствует, как и процедура останова — STOP. Кроме того, в алголе ЕС отсутствуют стандартные функции \arcsin и \lg , и нужно при переводе заменить их комбинацией имеющихся функций.

Другой трудностью является то, что набор символов в ЕС сильно ограничен по сравнению с ТА-1М. В частности в ТА-1М имеются строчные (малые) и заглавные латинские, русские, греческие буквы, а в ЕС — только заглавные латинские буквы. Напомним, что в ТА-1М для изображения греческих и заглавных букв при перфорации используются знаки \diamond и $-$. Например:

$v \sim V, V \sim -V, л \sim Л, Л \sim -Л, \sigma \sim \diamond S, \Sigma \sim -\diamond S$

Идентификаторы в алголе ЕС различаются по 6-и первым символам, и поэтому возможно «склеивание» различающихся в алголе ТА-1М идентификаторов.

Кодировка символов на перфокарте для М-220 является построчной, в отличие от построчной в ЕС ЭВМ.

Трудности перевода обращений к стандартным подпрограммам и необходимость «расклеивания» идентификаторов приводят к мысли

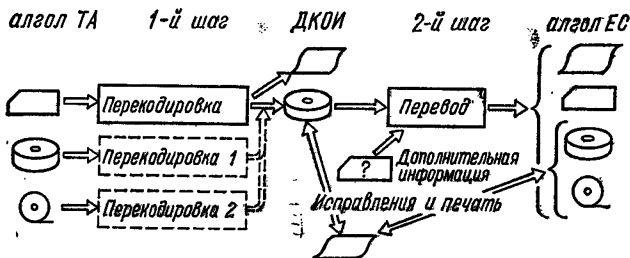


Рис. 17. Уточненная схема процесса перевода.

о целесообразности, по крайней мере на первом этапе реализации, полуавтоматического по своему характеру процесса перевода, требующего участия программиста (например, задания дополнительной информации, помогающей переводу, или даже ручного исправления и дписывания полученной программы на алголе ЕС).

Необходимо, кроме того, учитывать, что работать пользователю придется со старыми перфокартами, и желательно ограничиться лишь одноразовым их вводом для переписи во внешнюю память и для дальнейшей обработки, которую, может быть, придется проводить и неоднократно (учитывая возможный полуавтоматический характер перевода).

Проект. На рис. 17 приведена уточненная в результате анализа схема разрабатываемой программы ПЕРЕВАЛ (ПЕРЕВОД Алгола).

Перевод происходит за 2 шага: перекодировка и собственно перевод. На первом шаге происходит ввод перфокарт, содержащих алгольную программу ТА-1М в построчной кодировке, посимвольная перекодировка в код ДКОИ, принятый в ЕС ЭВМ, запись на магнитный диск и печать перекодированного текста. Для перекодировки программ ТА-1М, находящихся во внешней памяти (см. замечание к 6.2.1), могут потребоваться специальные программы перекодировки. Перекодировка осуществляется почти посимвольная с заменой отсутствующих в ДКОИ символов на один или несколько других в соответствии со следующей таблицей:

ТА-1М	×	10	√	∧	÷	↑	≤	≥	≠	'	'
Алгол ЕС	*	'	!	&	/'	**	<=	>=	7=	□	@
ТА-1М	=		⊃	◇	-	[]	-	ошибка		
Алгол ЕС	'EQUIV'		'IMPL'		G	O	()	"	#	

Большинству символов поставлены в соответствие те комбинации символов, которые имеются в алголе ЕС; буквы G и O выбраны в качестве признаков греческих и заглавных букв. Кавычка (') и знаки □ и @ будут до шага перевода соответственно обозначать начала ключевого слова и кавычки строки (такие временные знаки использованы для того, чтобы облегчить при последующем переводе обнаружение соответствующих конструкций в перекодированном тексте). Кроме того, русским буквам необходимо поставить в соответствие какие-то комбинации латинских букв (таблица выбранного соответствия здесь не приводится). Перекодировку русских букв придется осуществлять на стадии перевода, так как она не всегда является формальной: перекодировку нужно осуществлять только в идентификаторах, а в строках и комментариях алгола ее производить не нужно.

Чтобы можно было для исправления полученного текста в коде ДКОИ использовать служебную программу UPDATE (см. 3.2.1), перекодированные строки текста должны занимать не более 72 позиций в 80-байтных записях; неумещающиеся в такую запись символы программа перекодировки должна переносить в дополнительную запись (строчку).

На втором шаге производится собственно перевод, то есть замена сочетаний символов ТА-1М на группы символов алгола ЕС (ключевые слова, стандартные идентификаторы, функции, операторы и русские буквы). Функция $\text{tg}(x)$ заменяется на $\text{SIN}(x)/\text{COS}(x)$, а $\text{arcsin}(x)$ на $\text{ARCTAN}(x/\text{SQRT}(1-x*x))$ с выдачей предупреждающей диагностики. В случае невозможности или сомнительности перевода печатается предупреждающая диагностика; это относится, в основном, к переводу обращений к стандартным подпрограммам. В дополнительно вводимой информации задаются данные, предупреждающие склеивание идентификаторов алгола ЕС после их перевода, а также другие данные, необходимые для управления работой программы перевода.

Для того, чтобы была возможность транслировать переведенный текст в ОС ЕС ЭВМ, а также исправлять его по служебной программе UPDATE, записи во внешней памяти должны иметь длину 80 байтов, из которых занимают под текст не более 72 байтов; неумещающие в 72 байта символы образуют дополнительную запись (строчку).

З а м е ч а н и е 1. После перевода (как и после перекодировки) строчка текста может содержать более 72 символов и из неумещающих символов образуется дополнительная строчка. Если дополнительная строчка уже была образована при перекодировке, то при переводе лишние символы должны переноситься в нее. Признаком окончания строчки в алголе ТА-1М является комбинация символов `%|`, которая входит в число 72 символов после перекодировки. Эта комбинация сохраняется после перекодировки и уничтожается после перевода. Содержимое дополнительных строчек текста для большей наглядности перед распечаткой и выводом сдвигается в последние правые позиции.

З а м е ч а н и е 2. Расклеивание идентификаторов можно произвести уже после перевода с помощью препроцессорных средств (см. П. 4 и п. 3.2.2). Кроме того, поскольку выбор внешнего носителя для вывода переведенной программы может быть осуществлен с помощью директивы DD, дополнительная информация оказывается лишней и в рис. 17 может быть убрана.

Проект инструкции (для пользователя). Программа ПЕРЕВАЛ предназначена для перевода программ, написанных на алголе-60 для транслятора ТА-1М ЭВМ типа М-220, в программы на алголе-60 ЕС ЭВМ.

Основным вариантом является случай, когда программа хранится на перфокартах в кодировке устройства подготовки перфокарт (УПП) для машин типа М-220. Если переводимая программа находится на диске или ленте, должна быть составлена другая пере-

кодирующая программа, требования к которой будут сформулированы отдельно.

Перевод осуществляется следующими двумя шагами: перекодировка (PERECOD) и перевод в ЕС (PEREVEC). Шаги в задании для ОС ЕС имеют следующий вид:

```
//PERECOD EXEC PGM=PERECOD
//INTA      DD      характеристики-вводимого-НД
//OUTDISK DD      характеристики-НД-на-диске
//PRINT1 DD      SYSTOUT=A
//PEREVEC EXEC PGM=PEREVEC
//INDISK DD      характеристики-НД-на-диске
//PRINT2 DD      SYSOUT=A
//OUTEG DD      характеристики-выводимого-НД
```

Рекомендуется выполнять шаг перевода в другом задании, отдельно от первого шага, предварительно убедившись по распечатке в том, что ввод и перекодировка переводимой программы прошли правильно (таблица перекодировки приведена выше). Для трансляции полученной после перевода программы необходим еще один, третий шаг, в отдельном задании. Ввод с перфокарт производится образами карт с указанием параметра CODE=M в директиве DD с именем INTA; колода с перекодируемой программой, тем самым, не входит в текст задания для перекодировки. Длина записей для всех используемых наборов данных равна 80.

Текст программы после шага перекодировки или перевода занимает не более 72 байтов в записи, и поэтому в случае необходимости может быть исправлен с помощью служебной программы UPDATE (после его перенумерации). В текст переведенной программы вставляются диагностические или предупреждающие сообщения в случае, если перевод оказался невозможным или не гарантирующим правильность программы. В случае невозможности перевода, переведенный текст заменяется на знаки размыкания (подчеркивания), а сам текст печатается на следующей строчке в соответствующих позициях и во внешнюю память не записывается. Предупреждение имеет вид знаков размыкания, подчеркивающих сомнительный переведенный текст; знаки размыкания во внешнюю память не записываются. Например:

$$\text{STOP} \rightarrow \left\{ \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \end{array} \right. \text{STOP} \text{ и } \arcsin(d) \rightarrow \left\{ \begin{array}{c} \text{ARCTAN}(D/\text{SQRT}(1-D \cdot D)) \\ \text{---} \end{array} \right.$$

В результате перевода может произойти «склеивание» (совпадение) переведенных идентификаторов, поскольку в алголе ЕС они

различаются лишь по первым 6 символам; совпадающие идентификаторы обнаруживаются транслятором ЕС ЭВМ. Изменение одного из склеивающихся в результате перевода идентификаторов удобно производить, используя препроцессорные средства (см. пример в 3.2.2).

Общая схема процесса перевода представлена на рис. 17.

План тестирования. Тест для проверки программы перекодировки должен содержать все символы УПП, которые могут быть расположены в любом порядке. Необходимо проверить и случай, когда перекодированная строка теста (вместе со знаками | и %) не уместается в 72 символа. Следует также проверить возможность использования служебной программы UPDATE для исправления и печати информации, записанной на диск, с последующим переводом исправленного текста.

Для программы перевода в тестах проверяется перевод некоторых ключевых слов и всех переводимых и отдельных непереводимых обращений к стандартным процедурам (и функциям). С применением программы UPDATE тестируется возможность исправления и печати переведенной программы, записанной на диск или магнитную ленту, с последующей ее трансляцией. Кроме того, переводятся 1—2 небольшие реальные программы и производится их трансляция и счет в дальнейшей сверке полученных результатов; это осуществляется для программы, выдаваемой как во внешнюю память, так и на перфокарты.

6.2.3. Алгоритм. Процесс преобразования текста ранее был разбит на 2 этапа, реализуемых программами ПЕРЕКОД и ПЕРЕВЕС. Каждая программа обрабатывает информацию ограничений, на объем которой ранее наложено не было, и поэтому ее обработку следует производить отдельными порциями. Для программы перекодировки в качестве порции естественно взять информацию, содержащуюся на одной перфокарте. Для программы перевода порцией может быть и ключевое слово, и указатель (или идентификатор) стандартной функции, и весь оператор обращения к стандартной подпрограмме.

На рис. 18 приведены самые общие алгоритмы реализации этапов перекодировки и перевода. Здесь и ниже в алгоритмы, как и при реальной разработке, могут проникнуть ошибки, которые придется обнаруживать и удалять с помощью способов, изложенных ранее в книге.

Разработка алгоритма программы ПЕРЕКОД здесь не приводится ввиду его сравнительной простоты. Лишь оператор перекодировки требует дальнейшей проработки, а остальные операторы могут быть запрограммированы почти сразу (хотя придется учи-


```

ПЕРЕКОД: начало программы;
             пока не конец колоды цикла
             начало ввод перфокарты;
             перекодировка п/к;
             вывод на печать
             и запись на диск
             конец;
             окончание программы ПЕРЕКОД
ПЕРЕВЕС: начало программы;
             пока не конец текста цикла
             обработка переводимой порции;
             окончание программы ПЕРЕВЕС

```

Рис. 18. Общий алгоритм перевода.

тывать случай, когда перекодированная перфокарта занимает не одну, а две строчки печати или записи на диске).

Переводимые порции имеют достаточно разнообразное строение и требуют различных подходов к своему переводу. Можно предложить следующую классификацию переводимых порций.

Переводимая порция:

1. Ограничитель — перевод не требуется.
2. Ключевое слово — окаймляющие подчеркивание (после перекодировки — ‘’) и пробел заменяются на апострофы, а ключевое слово не изменяется (например, _END_ → ‘END’).
3. Идентификатор (может быть, с последующим списком параметров — аргументов).
 - 3.1. Нестандартные и стандартные функций, допустимые в алголе ЕС — перевод не требуется.
 - 3.2. Недопустимые стандартные функции (ARCTG, TG, ARCSIN) — замена и выдача предупреждающей диагностики.
 - 3.3. Подпрограммы с именами вида Рццц или STOP — для первой очереди программы производится выдача диагностики.
4. Строки — перевод не требуется.
5. Комментарии — перевод не требуется.

Таким образом, перед тем как осуществлять перевод, необходимо определить тип порции и, может быть, дополнительно вид идентификатора. Тип порции (кроме комментариев) определяется по ее первому символу, вид идентификатора — путем его анализа. Для обработки комментария нужно обнаружить ключевые слова *comment* или *end*, поэтому обработку комментариев можно совместить с обработкой ключевых слов. Поскольку в ТА-1М строки могут присутствовать только в обращении к стандартным подпро-

граммам, их обработку удобно совместить с обработкой стандартных подпрограмм.

1°. *Первый шаг алгоритмизации.*

<сим := ввод;>

обработка переводимой порции ~

*~ {если сим = _ то обработка ключевого слова и комментария
иначе если сим = буква то обработка
идентификаторной порции
иначе вывод (сим)}*

Здесь *ввод* и *вывод* — процедуры ввода очередного символа и вывода символа или символьного текста; *сим* — переменная хранящая текущий символ.

Поскольку конец комментария и идентификатора, видимо, придется определять по следующему введенному символу (см. 5.1.2), то оператор ввода первого символа следует вынести вне цикла.

2°. *Второй шаг детализации.*

обработка ключевого слова и комментария ~

*~ {обработка кл-слова;
обработка комментария
обработка кл-слова ~
~ {вывод (');
пересылка кл-слова на вывод;
вывод (')}}
обработка комментария ~
~ если (кл-слово = end или comment)
то пересылка комментария
обработка идентификаторной порции ~
~ {ввод идентификатора;
определение типа порции и перевод}*

3°. *Третий шаг детализации.*

пересылка кл-слова ~ *<здесь не приводится>*

пересылка комментария ~ *<здесь не приводится>*

ввод идентификатора ~

~ пока (сим = буква или цифра) цикл

{добавление переведенного сим к ид; сим := ввод}

Пока не определено, будет ли величина *ид* представлена массивом или строкой, и поэтому образование идентификатора из символов детально не конкретизируется. Русские буквы, встречающиеся в идентификаторе, должны быть переведены в латинские (см. 6.2.2).

определение типа порции и перевод ~

если ид = 'TG' то замена функции tg

иначе если ид = 'ARCTG'

то вывод ('ARCTAN')
иначе если $ид = 'ARCSIN'$

то замена функции \arcsin с предупреждением
иначе если $ид = 'Рццци'$

то диагн-вывод оператора

иначе если $ид = 'STOP'$

то диагн-вывод ('STOP')

иначе вывод ($ид$)

4°. Четвертый шаг.

замена $tg \sim \{ \text{ввод аргумента в } арг; \\ \text{вывод ('SIN(арг)/COS(арг'))} \}$

замена $\arcsin \sim \{ \text{ввод аргумента в } арг; \\ \text{вывод с предупреждением} \\ ('ARCTAN(арг/SQRT(1 - арг*арг))) \}$

диагн-вывод оператора $\sim \{ \text{ввод аргументов в } арг; \\ \text{диагн-вывод ('ид(арг'))} \}$

В последнем случае можно было бы просто переслать аргументы на вывод, но поскольку в будущем есть надежда, что обращение к стандартным подпрограммам удастся перевести, принято решение запомнить весь список аргументов для дальнейшего их разбора. На длины величин $арг$ и $ид$ придется наложить ограничения.

ввод аргумента (аргументов) в $арг \sim \langle \text{здесь не приводится} \rangle$

5°. Пятый шаг.

5.1°. $ввод \sim \text{начало цел } i \text{ статич нач (72),}$
статич симв $старый \text{ нач } (\square)$

$i := i + 1;$

если $i > 72$

то $\{ i := 1; \text{чтение записи } (c_i) \};$

если $(старый = \% \text{ и } c_i =)$

то $\{ старый := c_i; i := 72 \}$

иначе $старый := c_i;$

возврат (c_i)

конец

Переменная $старый$ введена для обнаружения сочетания, оканчивающего строчку (%). Переменная i — статическая (собственная, общая), она сохраняет свое значение после выхода из процедуры ввода. Служебное слово $нач$ определяет начальное значение для описываемой переменной, $цел$ — целый, $симв$ — символьный.

5.2°. Как установлено в проекте, определяются следующие 3 вида вывода: обычный, диагностический, предупреждающий, кото-

рые отличаются только присутствием и местом расположения диагностирующих знаков разделения (—). Алгоритмы выводов имеют общую структуру:

вывод (текст) ~ {l:= длина (текст);
 для m=1 до l цикл
 вывод символа (текст. сим_m)}

(текст.сим_m — m-й символ текста).

вывод символа ~

~ {если конец строки то {запись накопленных символов;
 очистка накапливающей величины};
 накопление очередного символа
 <может быть, с диагностикой>}

Дальнейшая проработка вывода символа здесь не приводится, но следует напомнить, что необходимо принять меры к тому, чтобы сочетание знаков % и | не попало в переведенный текст.

6°. Шестой шаг. Прежде чем приступить к компоновке модулей программы и их программированию, стоит просмотреть разрабатываемый алгоритм, чтобы убедиться, что в нем нет грубых ошибок или описок, а также каких-либо несоответствий проекту. Может оказаться полезным, минуя проект, обратить свое внимание на различные детали в постановке задачи, возможные особые случаи. В нашем примере, после внимательного изучения входного языка транслятора ТА-1М [16], обнаруживается, что в проекте и алгоритме не учтены, по крайней мере 2 момента:

1) ключевые слова могут задаваться в ТА-1М тремя первыми буквами;

2) стандартные идентификаторы необходимо сравнивать учитывая, что в них могут находиться пробелы.

Для исправления этих недоработок вернемся к шагу 3° (или 2°).

6.1°. пересылка к-слова ~ <здесь не приводится>

6.2°. Для исправления второй недоработки достаточно при вводе идентификатора получать наряду с обычным идентификатором (назовем его *ид-с-проб*) также и уплотненный идентификатор (*ид-плот*) — с удаленными пробелами.

ввод идентификатора ~

~ {ид-с-проб := пусто; ид-плот := пусто;
 пока (сим = буква или цифра или пробел)
 цикл {добавление сим к ид-с-проб;
 если сим ≠ | то добавление сим к ид-плот;
 сим := ввод}}

Везде, где в дальнейшем производится сравнение идентификатора со стандартным идентификатором, нужно вместо *ид* использовать *ид-плот*, а в остальных случаях — *ид-с-проб*.

6.3°. Пришла пора решить вопрос о форме представления таких величин, как идентификатор (*ид-с-проб* и *ид-плот*), список аргументов (*арг*), ключевое слово (*кл-слово*) и детализировать оператор добавления символа к этим величинам.

Примем решение представлять упомянутые величины в виде символьных строк (а не массивов). Тогда, например,

добавление *сим* к *кл-слову* ~

~ *кл-слово* := *кл-слово* || *сим*

Здесь используется операция сцепления строк. Причем, если *сим* является русской буквой, то в идентификаторе он должен заменяться на латинскую (используя функцию преобразования *руслат*), например:

добавление *сим* к *ид-плот* ~

~ если *сим* русский то *ид-плот* := *ид-плот* || *руслат(сим)*;
иначе *ид-плот* := *ид-плот* || *сим*;

Тоже и для *ид-с-проб*.

7°. *Седьмой шаг*. Уточним общую структуру нашей программы ПЕРЕВЕС представив ее в виде дерева модулей (см. рис. 19):

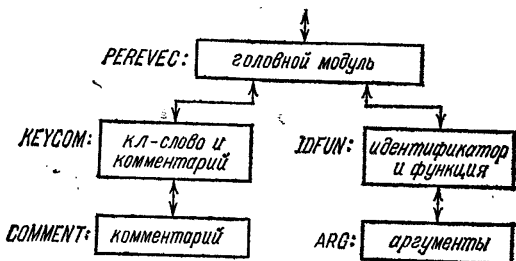


Рис. 19. Общая структура программы ПЕРЕВЕС.

Кроме того, каждый модуль связан с подпрограммами ввода и вывода: IN и OUT.

Перед тем, как перейти к программированию каждого модуля, имеет смысл собрать вместе все фрагменты алгоритма, которые относятся к этому модулю. Например, собранный алгоритм для головного модуля выглядит следующим образом:

PEREVES: <головной модуль>

начало печатать заголовка;

сим := IN;

пока не конец текста цикл
 если сим = подчеркивание то вызов KEYCOM
 иначе если сим = буква то вызов IDFUN
 иначе OUT (сим);
 конец цикла;
 печать об окончании работы
 конец

В ходе такого собирания алгоритмов модулей из разработанных ранее фрагментов (или после сборки, но до программирования) предстоит еще решать вопросы защиты от ошибок и способов реакции на них. Предстоит решить вопрос о том, какие ошибки будут преодолеваются в самом модуле, обнаружившем ошибку, а какие — в вышестоящих модулях или специальном модуле реакции на ошибки.

Кроме того, должны быть уточнены связи между модулями и определены формы такой связи (параметры-аргументы или общие величины). Например, так как конец текста придется устанавливать по его исчерпанию в модуле IN, встает вопрос о передаче сведений об этом факте в головной модуль. Можно, конечно, пересмотреть несколько схему головного модуля и оканчивать работу (нормально или аварийно) в специальном модуле, куда и обращаться из модуля IN.

Если в ходе сборки алгоритма модуля или при его программировании окажется, что он не уместается в установленные размеры, то нужно один (или несколько) фрагментов оформить в виде модуля.

8°. *Тестирование.* Для каждого модуля нужно запроектировать алгоритмические тесты, проверяющие все его ветви или операторы.

При тестировании головного модуля можно прибегнуть и к нисходящему тестированию, обращаясь к имитаторам четырех вызываемых модулей. Имитаторы IN и OUT могут быть аналогичны имитаторам INSYM и OUTSYM, приведенным в 6.1.3, но ввод и вывод записей (перфокарт) лучше производить в них самих и со ссылками на директивы DD с именами INDISK, OUTEC, PRINT2. Имитатор KEYCOD выбирает по IN и выдает по OUT все символы до первого пробела. Если ограничиться только однобуквенными идентификаторами в тестах, то имитатор IDFUN должен просто печатать полученный символ. Если вадать на вход модуля IN следующий текст:

A:= _IF X < 0 _THEN 1.2*(Y+Z) _ELSE R(T)

то на выходе из OUT мы должны получить его же,

Если программа ПЕРЕКОД уже готова, то задание должно иметь вид, приведенный в 6.2.2, а текст должен быть отперфорирован в коде ТА-1М. В противном случае, текст может быть изготовлен в коде ДКОИ для ЕС ЭВМ и подложен к (единственному) шагу PEREVEC, в котором характеристики-НД-на-диске заменяются на звездочку. Текст можно записать и на диск в соответствующий набор, с помощью имитатора PERECOD.

Нисходящее тестирование позволит нам заранее проверить и головной модуль, и задание, спроектированное ранее. В дальнейшем использованные имитаторы будут постепенно заменяться на реальные модули, по мере их изготовления и автономной отладки.

П Р И Л О Ж Е Н И Е

ПРЕПРОЦЕССОРНЫЕ СРЕДСТВА

П.1. Общие сведения

Препроцессорные средства предназначены для облегчения создания («генерации») программы на некотором алгоритмическом языке. Такие средства называют также макросредствами, средствами макрогенерации, средствами времени компиляции (трансляции). Примеры применения препроцессорных средств, предупреждающих появление многих ошибок программирования, были даны в гл. 4 и 5. Кроме того, в гл. 1, 2 и 3 приводились примеры использования этих средств на этапах локализации и исправления ошибок в отлаживаемой программе.

Ввиду широкого использования препроцессорных средств в современных методах разработки программ и частых ссылок на них из многих мест этой книги, в настоящей главе дается описание препроцессорных средств, имеющихся в системе программирования ОС ЕС ЭВМ.

Препроцессорные средства являются частью транслятора — они обрабатывают текст транслируемой программы непосредственно перед трансляцией, называемой процессорной обработкой. То есть, можно сказать, что трансляция программы проходит две стадии: *препроцессорную* и *процессорную*, а транслятор состоит из двух частей — *препроцессора* и *процессора*. На первой стадии с помощью препроцессорных средств из препроцессорной программы, содержащей препроцессорные операторы, получается процессорная программа, являющаяся исходной для следующего этапа, на котором и производится собственно трансляция полученной только что программы (см. рис. 20); вместо трансляции полученная программа может быть выдана на внешний носитель (перфокарты, диски, ленты). Разумеется, препроцессорный этап может и отсутствовать, тогда исходная процессорная программа поступает на трансляцию с перфокарт или из внешней памяти, а не от препроцессора.

Основными видами работ по генерации текста программ, выполняемыми препроцессорными средствами, являются следующие: вставка заранее заготовленного текста в указанное место программы с одновременным проведением необходимых систематических модификаций в тексте; изменение порядка следования фрагментов

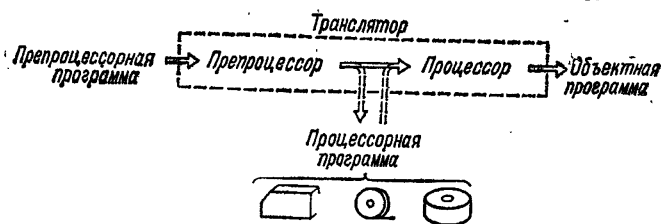


Рис. 20. Две стадии трансляции.

текста, подготовленного к трансляции (обход или повторение фрагментов текста); замена имен, содержащихся в программе, на произвольный текст; генерация фрагментов текста программы по заданным правилам (процедурам).

Хотя рассматриваемые препроцессорные средства предназначены для генерации программ на языке PL/1 и включены в транслятор с этого языка, они могут быть с успехом использованы и для других алгоритмических языков, на которых программисты работают в ОС ЕС ЭВМ, в частности, это касается алгола-60 и фортрана-IV ЕС. Ограничения при этом касаются только использования русских букв в строках и комментариях, которые приходится оформлять по новым правилам.

Ниже, при изложении препроцессорных средств PL/1, обязательного знакомства читателя с языком PL/1 не предполагается. Но читатель должен иметь представление об одном из трех названных языков, об основных его понятиях, таких как переменные, функции, операторы, описания. Для облегчения изучения препроцессорных средств читателями, не знакомыми с PL/1, изложение средств ведется в неполном объеме. Для тех, кто знаком с PL/1, заметим, что язык препроцессорных средств является очень ограниченным и специфическим его подмножеством (см. П.9).

В конце раздела П.7 приводятся типовые задания, обеспечивающие применение препроцессорных средств как для программ на PL/1, так и на фортране и алголе. Кроме того, приведены синтаксические формы изложенных препроцессорных средств.

П.2. Пример

```
REAL A(10), B(10), C(10)
. . . . .
```

```

% DECLARE I FIXED, X CHARACTER;
% I = 1; % X = '1.234 * B';
% MET: ;
    A (I) = X (I) + C (I)
% I = I + 1;
% IF I < 5 % THEN % GO TO MET;
. . . . .
END

```

Приведенный текст содержит как операторы фортрана, так и препроцессорные операторы (начинающиеся со знака процента и оканчивающиеся точкой с запятой), и поэтому может рассматриваться как пример препроцессорной программы (многоточия заменяют произвольные операторы фортрана). Выполнение этой программы препроцессором происходит следующим образом.

Оператор описания фортрана, не принадлежащий к препроцессорным операторам, будет без изменений передан на выход из препроцессора для последующей трансляции или, может быть, для предварительного запоминания на внешних носителях (см. рис. 20). Таким же образом обрабатываются и другие операторы фортрана, отмеченные многоточием и предшествующие первому препроцессорному оператору.

Препроцессорный оператор описания объявляет имена I и X, как препроцессорные переменные целого («фиксированного») и символьного типа. Следующие препроцессорные операторы присваивают объявленным переменным указанные значения, выраженные числовой и символьной (строчной) константами. Пустой оператор, не выполняющий никаких действий, служит лишь для помещения перед ним препроцессорной метки, которая пригодится нам в дальнейшем.

В операторе фортрана, следующем далее в препроцессорной программе, содержатся обе препроцессорные переменные I и X, и перед передачей транслятору текст оператора изменяется в соответствии с текущими значениями этих препроцессорных переменных. Поэтому передаваемый на трансляцию оператор фортрана приобретает следующий вид:

$$A(I) = 1.234 * B(I) + C(I)$$

Конечно, на месте оператора фортрана мог быть и аналогичный или какой-либо другой оператор PL/I или алгола ЕС, поскольку в препроцессоре проверка правильности синтаксиса исходного и полученного операторов не производится.

Следующие препроцессорные операторы в программе увеличивают значение препроцессорной переменной I на единицу и сравнивают полученное значение с 5. Поскольку значение I меньше 5,

происходит переход на оператор, помеченный меткой MET. Оператор является пустым и далее производится обработка текста, следующего за пустым оператором.

Так как $I=2$, а значение X не изменяется, то передаваемый для последующей трансляции текст будет иметь вид:

$$A(2)=1.234*B(2)+C(2)$$

Подобные операторы будут отсылаться на трансляцию до тех пор пока $I \leq 4$, то есть последний из них, четвертый оператор получит следующий вид:

$$A(4)=1.234*B(4)+C(4)$$

Последним переданным из препроцессора оператором фортрана будет оператор END, после чего препроцессорная обработка для данного примера заканчивается, и транслятор может приступить к трансляции следующей программы фортрана, полученной на выходе препроцессора:

```
REAL A(10), B(10), C(10)
. . . . .
A(1)=1.234*B(1)+C(1)
A(2)=1.234*B(2)+C(2)
A(3)=1.234*B(3)+C(3)
A(4)=1.234*B(4)+C(4)
. . . . .
END
```

Как видим, препроцессорные операторы в сгенерированный текст не входят.

Теперь, после общего знакомства с препроцессорными средствами, мы приступаем к систематическому изучению основных понятий и возможностей этих средств. Терминология будет использоваться та, которая принята в PL/1, но, кроме того, приводятся и эквивалентные термины, свойственные фортрану и алголу. Помимо примеров, приводимых ниже, ряд более содержательных примеров имеется в основных главах, куда и даются ссылки.

П.3. Основные элементы и выражения

Символы. Основные символы, которые служат для представления конструкций препроцессорных средств, можно разделить на буквы, цифры и спецзнаки. Кроме заглавных букв латинского алфавита используются и три дополнительные «буквы»: \square , @, #. Цифры применяются десятичные. Спецзнаки будут перечислены в ходе последующего изложения (практически они совпадают с набором

спецзнаков PL/1). В некоторых специальных конструкциях допускается использование всех символов, имеющихся на внешних устройствах конкретной ЭВМ — *символов ЭВМ*, включающих в себя, в частности, русские буквы и основные символы.

Идентификаторы и метки. Идентификаторы (имена) представляют собой последовательность из букв, цифр и знака разделения (—), начинающуюся обязательно с буквы и содержащую не более 31 символа. Пробелы не могут помещаться внутри идентификатора.

П р и м е р ы.

V V10
VELOCITY_OF_WIND

Идентификаторы используются для обозначения имен препроцессорных переменных, функций, а также для изображения меток препроцессорных операторов; в последнем случае они присоединяются к оператору с помощью двоеточия.

П р и м е р ы м е т о к.

М: BACK:

З а м е ч а н и е. Препроцессорные идентификаторы могут иметь вид и стандартных идентификаторов, используемых в каком-либо алгоритмическом языке, таких, например, как имена стандартных функций, ключевые слова, форматы и т. п. Например: SIN, IF, CALL, B, E и т. д.

Константы. Препроцессорные константы могут быть трех видов: арифметические (числовые), символьные (литеральные) и битовые (логические).

Арифметическая константа, или просто *число*, представляет собой целое десятичное число, может быть, со знаком, состоящее не более чем из 5 цифр. Пробелы внутри константы не допускаются.

Символьная константа — *строка* — имеет вид последовательности символов ЭВМ, заключенной в апострофы; внутренние апострофы при записи строки удваиваются. Строки могут располагаться на нескольких строчках бланка, и длина их практически неограничена. Пробел является одним из символов строки, и его можно для ясности обозначать при записи знаком \square .

Битовая константа может иметь вид '1'B или '0'B, что соответствует логическим (булевским) константам .TRUE. или .FALSE., true или false в фортране и в алголе соответственно. Пробелы внутри констант не допускаются.

П р и м е р ы.

—135 +98765 0 0010
'A * 3.14' 'REPLACE" A \square + \square B"

Переменные. Среди понятий препроцессорных средств отсутствуют массивы и переменные с индексами, и поэтому препроцессорные переменные представляются просто их именами (идентификаторами). По типу принимаемых значений переменные делятся на арифметические и символьные. Значения арифметических переменных представляются числами (арифметическими константами). Значением символьной переменной является символьная строка без внешних апострофов и с заменой каждой пары внутренних апострофов (см. константы) на один апостроф. Количество цифр в арифметическом значении и количество символов в строчном значении определяют соответственно *точность* и *длину*. Точность или длина значения переменной определяются той точностью или длиной, которую имело последнее значение, присвоенное данной переменной. Например, если арифметической и символьной переменным присвоить (об операторах присваивания будет сказано ниже) приведенные выше константы, то значения, полученные при этом переменными, будут иметь следующий вид:

—135 +98765 0 0010 (точность: 3,5,1,4);
 A*3.14 REPLACE'A□+□B' (длина: 6 и 14)

Ограничение. Значения строчных переменных, которые входят в текст обычных, не препроцессорных операторов, как правило, должны содержать четное число апострофов. Например, следующая строчная константа, представляющая значение одной из таких препроцессорных переменных:

'REPLACE""ON''

недопустима, так как ее значение

REPLACE''ON'

содержит 3 апострофа.

Описание переменных. Каждая препроцессорная переменная, до того как она в первый раз используется в программе, должна быть описана (специфицирована). Для каждой переменной с помощью описателя (атрибута) FIXED или CHARACTER (сокращенно CHAR) должен быть указан тип принимаемых ею значений: арифметический («фиксированный») или символьный. Описание (оператор описания) может иметь, например, следующий вид (см. также пример, приведенный выше в П. 2):

%DECLARE A FIXED, R CHARACTER,
 (C, N, F) CHAR;

Как видим, общие описатели могут выноситься за скобки. Одно описание может содержать практически любое количество идентификаторов и описателей.

Функции. Функции по типу вычисляемого значения разделяются на арифметические и символьные. Аргументами функции могут быть только препроцессорные арифметические и символьно-строчные константы и переменные. Вычисленное значение замещает собой обращение к функции (указатель функции) в теле препроцессорной программы. Особенности задания аргументов при обращении к препроцессорным функциям и подробное описание действий, производимых при этом, будут изложены ниже (см. П.5).

Имеется только одна встроенная (стандартная) препроцессорная функция — SUBSTR, обращение к которой имеет вид: SUBSTR(x, m, n). Функция из символьной препроцессорной строки x (константы, переменной, выражения), начиная с m -го символа, выделяет подстроку длины n (то есть, кончая символом с номером $m+n-1$); m и n — препроцессорные выражения. Например,

SUBSTR (FOR, LOW, 3) = 'N — 8'

для случая, когда

FOR = 'M*(N-8)' и LOW=4.

Апострофы здесь используются для указания на то, что значение имеет символьный тип; они не являются частью значения.

Выражения. Препроцессорные выражения по типу вычисляемого результата могут быть разделены на арифметические, символьные и логические (битовые). Тип результата определяется по типу операции, выполняемой над операндами: операции $+$, $-$, $*$, $/$ являются арифметическими, операция сцепления строк (изображаемая двумя восклицательными знаками — $!!$) — символьной, операции сравнения ($<$, $<=$, $>$, $>=$, $=$, $\neg=$), а также операции \neg (не), $\&$ (и), $!$ (или) — логическими. Операция возведения в степень отсутствует. В выражениях могут использоваться также и круглые скобки.

В препроцессоре результаты арифметических операций всегда представляются в виде целых чисел, например: $3/5=0$ и $5/3=1$.

Операция сцепления соединяет («сцепляет») значения символьных операндов, добавляя справа к первому операнду второй; длина результата равна сумме длин операндов. Например, для переменной FOR, определенной выше, получим, что

FOR $!!$ + TOW = 'M*(N-8)+TOW'

Операции сравнения могут производиться как над числовыми данными, так и над символьными. В последнем случае сравнение строк производится посимвольно слева направо; значения символов определяются в соответствии с кодировкой ДКОИ; перед сравнением более короткая строка дополняется справа пробелами.

Примеры операций:

I <= 10 SYM \neg = 'Z' X > Y $!!$ Z

Предупреждение. Операция логического отрицания считается самой старшей в выражении. Операция сцепления выполняется после арифметических операций, но до операций сравнения.

Преобразования. В случае выполнения над арифметическими данными символьной операции (!!), операнды преобразуются к символьной строке длины 8, путем добавления необходимого количества пробелов перед старшей значащей цифрой или знаком минус (плюс опускается) значения, каждая цифра которого представлена в виде символа. Например, для $F = '+KIN'$ получаем следующий результат:

$$-0185 !!F = '_ _ _ _ -185 + KIN'$$

Такое же преобразование может производиться и в других случаях, например, при присваивании или выделении подстроки.

Препроцессор осуществляет и другие типы преобразований, аналогичные тем, которые имеются в PL/1, но они здесь не рассматриваются.

Пробелы и комментарии. В отличие от алгола и фортрана в PL/1 и его препроцессоре использование пробелов подчиняется строгим правилам. Пробелы обязательно должны разделять числа, строки, идентификаторы и ключевые слова операторов. Перед спецзнаками или после них можно располагать один или несколько пробелов, а также комментарии, которые имеют следующий вид:

/ последовательность-символов-ЭВМ */*

Внутри комментария не должно быть сочетания **/*.

Пример комментария в препроцессорном выражении:

$$A + B * C - R * DED /* ФУНКЦИЯ */ + 1.49$$

Комментарии, которые входят в какой-нибудь препроцессорный оператор, в выходной текст препроцессора не попадают; в противном случае они передаются на выход без изменений.

П.4. Основные операторы

Действия, непосредственно осуществляющие генерацию программы, производятся препроцессорными операторами. Одни операторы служат для непосредственной генерации текста программы (операторы присваивания, вызова процедуры, включения), другие помогают выполнять такие действия.

Признаком препроцессорного оператора служит знак процента (%), который ставится перед оператором, может быть, отделяясь от него пробелом (пробелами); если оператор помечен, то % помещается перед меткой. Все операторы ограничиваются справа точкой с запятой,

Оператор присваивания. Оператор присваивания служит для присваивания значения выражения препроцессорной переменной. Например,

% $V1 = '(5 + T)'$; или **%** $МЕТКА: POL = POL + 1$;

Оператор присваивания является одним из основных средств, используемых для внесения изменений в текст генерируемой программы. Например, после выполнения указанных выше операторов, если при дальнейшей обработке в тексте встретятся идентификаторы $V1$ или POL , то они будут заменены на текущие значения препроцессорных переменных, то есть на те значения, которые они получили при последнем присваивании. Впрочем, имя POL может и не входить в текст генерируемой программы, а использоваться просто в качестве счетчика для управления выполнением препроцессорных операторов в коде генерации.

Особенность. Значение, заменяющее какую-либо препроцессорную переменную в тексте генерируемой программы, всегда отделяется от соседнего текста справа и слева одним пробелом. Кроме того, если переменная имеет арифметический тип, то перед заменой производится преобразование арифметического значения к виду символьной строки с длиной 8 (см. выше преобразования). То есть, например, если $V1$ и POL встретились при дальнейшей обработке программы в таком контексте:

... + R - $V1$ ** ($POL - 2.35$) *...

то для $POL = 6$ сгенерируется следующий текст:

... + R - $\square(5 + T)\square$ ** ($\square\square\square\square\square\square\square\square 6\square - 2.35$) *...

Операторы активации и дезактивации. Для того чтобы действительно происходили замены препроцессорных переменных в тексте генерируемой программы, эти переменные должны быть активны к моменту препроцессорной обработки фрагмента текста, содержащего имена заменяемых переменных. Активация замены выполняется оператором активации, который может иметь, например, следующий вид:

% $ACTIVATE V1, Z, POL$;

В списке вслед за ключевым словом **ACTIVATE** перечисляются активизируемые препроцессорные переменные. Первая активация производится оператором описания соответствующей препроцессорной переменной.

Отменить реализацию замен в тексте программы можно оператором дезактивации, который имеет вид, аналогичный оператору

активации:

```
% DEACTIVATE V1, Z, POB;
```

Значение дезактивированной переменной не пропадает, оно сохраняется до следующего оператора активации этой переменной.

Пример.

```
% DECLARE A FIXED, S CHAR;
```

```
% A=175;
```

```
% DEACTIVATE A;
```

```
% S='Y (K)';
```

```
T=A+S**4.5;
```

```
% ACTIVATE A; % DEACTIVATE S;
```

```
T=A+S/3.7E-4;
```

В результате обработки приведенного текста, на выходе препроцессора получим следующие операторы PL/1 (если не учитывать вставляемых пробелов):

```
T=A+Y(K)**4.5;
```

```
T=175+S/3.7E-4;
```

З а м е ч а н и я. 1. Операторы активации и дезактивации полагаются всегда после описания соответствующих переменных.

2. Операторы активации и дезактивации используются также и для имен процедур, о чем рассказано ниже (см. П. 5).

Оператор перехода. Для управления ходом препроцессорной обработки используется оператор перехода, пример использования которого был дан ранее (см. П. 2). Следующим после препроцессорного оператора перехода выполняется препроцессорный оператор, помеченный меткой, содержащейся в операторе перехода.

Пустой оператор. Пустой оператор служит обычно для помещения перед ним необходимой метки. Например:

```
% TUT;
```

В сочетании с оператором перехода пустой оператор может быть использован для обхода части транслируемого текста. Например, препроцессор, обработав следующий фрагмент:

```
101 X = P + 1
```

```
% GO TO TUT;
```

```
102 Y = P + 2
```

```
% TUT;
```

```
103 Z = P + 3
```

подготовит к предстоящей трансляции такие операторы фортрана:

```
101 X=P+1
```

```
103 Z=P+3
```

Пустой оператор используется и для помещения в него комментариев, которые не должны появляться в выходном тексте препроцессора. Например:

```
% /* ИЗМЕНЕНИЕ % ПЕРЕМЕННОЙ T */;
```

Другие примеры применения пустого оператора можно найти в П. 2 и 2.1.1.

Условный оператор. Условный препроцессорный оператор используется для разветвления процесса генерации программы и имеет следующее строение:

```
% IF логическое-выражение % THEN оператор  
% ELSE оператор
```

Оператор после % THEN выполняется, если логическое выражение истинно; в противном случае выполняется оператор, стоящий после % ELSE. Конструкция с % ELSE может и отсутствовать.

Примеры.

```
1) % IF PR = '.' % THEN % T = Q - 5;  
% ELSE % T = 0;  
2) % IF NOW = 1 % THEN % GO TO TUT;
```

Второй пример, в сочетании с приведенным ранее примером, демонстрирует вариант условного обхода текста, исключаемого из программы; см. также 2.1.1, Б. После % THEN и % ELSE могут находиться любые препроцессорные операторы, кроме операторов описаний (% DECLARE и % PROCEDURE).

Особенность. В препроцессорном условном операторе разрешается (аналогично PL/1) после % THEN и % ELSE вновь использовать оператор % IF. Но для случая неполного условного оператора (без % ELSE) такая вложенность операторов % IF приводит к логическим трудностям. Поэтому читателям, незнакомым с PL/1, следует избегать вложенных % IF без % ELSE (особенно работающим на фортране) или ограничивать их использование (например, как в условном операторе алгола).

Составной оператор. Составной (групповой) оператор является важным частным случаем оператора цикла, излагаемого ниже. Составной оператор позволяет с помощью «операторных скобок» % DO; и % END; объединить в один оператор несколько подряд идущих операторов и поместить их в условный оператор после % THEN или % ELSE. Например:

```
% IF PR = '.' % THEN % DO; % T = Q - 5;  
% GO TO L1; % END;  
% ELSE % T = 0;
```

В случае истинности логического условия, выполняются сразу два оператора, входящие в составной; отсутствие %DO и %END; привело бы к синтаксической ошибке, поскольку после %THEN может находиться только один оператор.

Составной оператор может содержать, наряду с препроцессорными операторами, также операторы и описания других языков (или их фрагменты), то что дальше мы будем называть словом «текст».

Пример.

```
D = % IF R < 0 % THEN % DO; A + B * C % END;
      % ELSE % DO; B + C * A % END;
      % /* D = A + B * C ИЛИ D = B + C * A */;
```

Таким образом, составной оператор в сочетании с условным позволяет включать в транслируемый текст то один, то другой фрагмент; такими фрагментами могут быть, например, операторы отладочной печати (см. 2.1.2, Б).

Оператор цикла. Оператор цикла можно представить в следующем виде:

```
% DO v = e1 TO e2 BY e3
                                операторы и(или) текст % END;
```

Здесь:

DO, BY, TO, END — ключевые слова, имеющие смысл:

ВЫПОЛНИТЬ, ШАГ, ДО, КОНЕЦ;

v — препроцессорная переменная, параметр цикла;

e₁, e₂, e₃ — арифметические выражения, задающие характеристики цикла.

Частицы TO и BY с выражениями могут быть переставлены местами; кроме того, BY с e₃ может отсутствовать — тогда шаг полагается равным единице (как в фортране).

Среди операторов, выполняемых в цикле, вместе с препроцессорными операторами могут находиться и операторы языка, на котором производится программирование, или произвольный текст. Препроцессорные операторы, входящие в тело цикла, многократно выполняются, а текст (может быть, и изменяемый в цикле) многократно вставляется в генерируемую программу.

Пример. (Ср. с примером в П. 2).

```
% DECLARE K FIXED;
% DO K = 1 BY 1 TO 3;
      A(/K/) := X(/K/) + C(/K/);
% END;
% DEACTIVATE K;
```

В генерируемой программе получим следующие операторы
алгола ЕС:

```
A(/1/) := X(/1/) + C(/1/);  
A(/2/) := X(/2/) + C(/2/);  
A(/3/) := X(/3/) + C(/3/);
```

Использование оператора дезактивации для переменной К позволяет программисту не волноваться по поводу наличия такого же идентификатора в других частях текста генерируемой программы. В противном случае, идентификатор К, который в последующем тексте, может быть, и не мыслился как препроцессорный, будет везде в дальнейшем тексте программы заменен на 4 (значение параметра цикла после исчерпания).

З а м е ч а н и е. После % END (как и в PL/1) может быть задана метка, отсылающая к метке перед оператором % DO, и связывающая, таким образом, в пару % DO и % END. Такая связь позволяет сделать запись программы более наглядной, особенно для случаев вложенных % DO. Кроме того, если одно из внутренних % END по ошибке пропущено, оно будет автоматически добавлено непосредственно перед специфицированным % END. Например, при генерации (программы на PL/1) может быть задано:

```
% DECLARE (I,J,M,N) FIXED;  
.....  
% ACTIVATE I, J, M, N;  
    % M=5; % N=12;  
% CYCL: DO I=1 TO N;  
    % DO J=1 TO M;  
        A(I,J)=0;  
    % END CYCL;  
% DEACTIVATE I,J,M,N;
```

То же справедливо и для составных операторов.

П.5. Процедура и обращение к ней

Основное назначение препроцессорных процедур-функций то же, что и в рассматриваемых алгоритмических языках, но имеются особенности при их использовании, поскольку обращение к ним производится не только из препроцессорных операторов, но и из текста алгоритмической программы.

Аппарат работы с препроцессорной процедурой-функцией состоит из трех основных компонент: описания процедуры, описания ее имени входа и обращения к ней.

Описание процедуры. Описанием процедуры, оператором процедуры или просто процедурой здесь называется то, что в фортране является подпрограммой-функцией, а в алголе — описанием процедуры-функции. Процедура состоит из заголовка, тела и оператора конца.

Заголовок имеет вид:

имя: PROCEDURE (список-параметров) RETURNS (описатель);

Имя процедуры задается в виде метки, но таковой здесь не является. В качестве описателя (атрибута) указывается FIXED или CHAR: он характеризует тип функции, то есть тип возвращаемого значения. Каждый параметр процедуры должен быть описан в теле процедуры; список параметров, заключенный в скобки, может и отсутствовать.

Тело процедуры содержит только препроцессорные операторы (и описания), причем знак процента перед ними опускается (!). Операторы % ACTIVATE; % DEACTIVATE, а также % PROCEDURE и % INCLUDE запрещаются. По крайней мере одним из операторов должен быть оператор:

RETURN (*выражение*);

который задает возвращаемое значение. Выход из процедуры по % GO TO запрещается.

Если в процедуре описываются переменные, имена которых совпадают с препроцессорными переменными вне процедуры, то такие переменные считаются разными, с различными сферами действия: внутри и вне процедуры. Если такого описания в процедуре нет, то переменные считаются тождественными.

Оканчивается процедура *оператором конца*, в котором для удобства может быть указано и имя оканчивающейся процедуры:

% END или % END имя;

Пример:

```
% FOUR: PROCEDURE (P, Z, Q) RETURNS (CHAR);  
      DECLARE (P, Z, Q) CHAR;  
      RETURN (('HPIIZIIQIIZII QUIZIIQIIZIIQI'));  
% END FOUR;
```

Содержательный смысл этой процедуры будет ясен позднее, при рассмотрении обращения к ней.

Описание имени входа. Перед тем, как произвести обращение к процедуре, необходимо описать ее имя как имя входа с помощью оператора % DECLARE, например, так (см. также процедуру FOUR):

% DECLARE FOUR ENTRY (CHAR, CHAR, CHAR) RETURNS
CHAR);

Как видим, описание входа заключается в том, что вслед за ключевым словом ENTRY дается в скобках список описателей для параметров, и далее приводится описание возвращаемого значения. Эта информация, по существу, дублирует описание, имеющееся в процедуре, но оно является обязательным.

Выполнение оператора, описывающего вход, активирует осуществление замен, производимых операторами вызова процедур (см. ниже), поэтому его следует располагать до первого вызова процедуры, который должен производить замену, и до операторов активации и деактивации (в операторах активации и деактивации указываются имена соответствующих процедур).

Процедуру следует располагать в препроцессорной программе до описания ее входа; описание входа располагается всегда вне тела какой-либо препроцессорной процедуры.

Обращение к процедуре. Обращение к процедуре может производиться как из препроцессорного оператора, так и прямо из обычного текста, содержащегося в обрабатываемой программе. В обоих случаях обращение имеет один и тот же вид, совпадающий с обращением к функции из выражения, и в обоих случаях обращение заменяется на значение, возвращаемое процедурой. Особенностью обращения из текста является то, что аргументы, являющиеся символьными константами, задаются своим значением, то есть без внешних апострофов и без удваивания внутренних апострофов.

П р и м е р:

```
BLOK: BEGIN;
      DECLARE (A, B, C) FLOAT;
      . . . . .
      C=FOUR (A, +, B)—FOUR (A, *, B);
      . . . . .
END BLOK;
```

Оператор присваивания в программе, написанной на PL/1, после генерации примет вид (см. выше описание процедуры-функции FOUR):

$$C = (A+B+B+B+B) - (A * B * B * B * B);$$

На аргументы препроцессорных функций, обращение к которым помещено в текст вне препроцессорных операторов, должны быть наложены ограничения, поскольку единственными признаками, ограничивающими аргументы, являются круглые скобки или запятые. Поэтому аргументы, являющиеся символьными константами, должны иметь сбалансированное количество скобок и все запятые должны находиться внутри них. Например, для указателя функции:

```
FOUR (G (A, B), *, (A + B))
```

аргументами являются символьные значения, представляемые следующими строками:

'G(A,B)', '*', '(A+B)'.

В результате обращения этот указатель функции заменится на следующее символьное значение:

G (A, B)*(A+B)*(A+B)*(A+B)*(A+B)

Если обращение входит в препроцессорный оператор (например, в оператор присваивания), то оно должно иметь следующий вид:

% E=FOUR ('G(A, B)', '*', '(A+B)');

Передача аргументов для функций, находящихся вне препроцессорных операторов, производится всегда их значением (то есть с использованием фиктивных аргументов); передача имени аргумента, позволяющая изменять значение аргумента-переменной, внешней по отношению к вызываемой процедуре, возможна только из препроцессорных операторов.

Все сказанное относится и к препроцессорной встроенной функции SUBSTR. Если обращение к SUBSTR производится из текста, то эта функция должна быть предварительно активирована.

З а м е ч а н и я. 1. Как и для препроцессорной переменной значение препроцессорной процедуры-функции, заменяющее обращение к функции вне препроцессорного оператора, окаймляется пробелами (см. П. 4).

2. Максимальное количество аргументов и, соответственно, параметров — 15.

П.6. Оператор включения текста

Оператор включения. Оператор включения предназначен для внесения в генерируемую программу текста, хранящегося на внешнем носителе (на магнитном диске), с одновременным проведением систематических изменений во включаемом тексте. Оператор имеет вид:

% INCLUDE имя-DD(имя-раздела);

Таким образом, оператор % INCLUDE включает в генерируемую программу раздел библиотечного набора данных.

Например, по оператору

% INCLUDE LINALG (TRANSP);

и директиве

//PLIL,LINALG_DD DSNAME=ALGEBR,

// VOL=SER=MATLIB,DISP=OLD

производится включение текста, находящегося в разделе TRANSP библиотеки ALGEBR.

В операторе можно указать сразу несколько разделов (вместе с именами DD), отделяя их при этом запятыми. Имя DD можно не указывать, если его взять совпадающим с SYSLIB.

Например, оператор

```
% INCLUDE LINALG(TRANSP), TEPLO,  
MOST, POLY (DIFF);
```

включает сразу 4 раздела (TRANSP, TEPLO, MOST, DIFF) из трех разных библиотек.

Включаемый текст. Если во включаемом тексте имеются препроекторные переменные, описанные в основной вызывающей программе, то они при включении заменяются на текущие значения этих препроекторных переменных (пример можно найти в п. 4.2.4.). Описание переменных во включенном разделе активирует замены в последующем тексте вызывающей программы.

Во включаемом тексте могут присутствовать различные препроекторные операторы, в том числе и % INCLUDE. Возможно и обращение из основного текста к включаемой процедуре, но только после ее включения в основной текст. Переход по оператору % GO TO допускается только из включаемого текста в основной или в уже включенный, но не наоборот.

Оператор % INCLUDE не может присутствовать в процедуре.

П.7. Программа и задание

Программа и ее обработка. Препроекторная программа, то есть информация поступающая на вход препроектора, представляет собой свободное сочетание препроекторных операторов с текстом программы на одном из алгоритмических языков: PL/1, фор-тране-IV, алголе-60 для ЕС ЭВМ (или в более общем случае — на произвольном языке, отвечающем некоторым требованиям). Текст может находиться и внутри некоторых препроекторных операторов (составного и цикла). Внутри текста конструкции, имеющие вид идентификатора или идентификатора с последующей парой скобок, могут быть объявлены препроекторными переменными и функциями, которые, тем самым, в ходе выполнения препроекторной программы будут заменены на произвольный текст, не содержащий препроекторных операторов. Каких-либо специальных операторов, отличающих начало и конец препроекторной программы, нет. Каждая препроекторная программа обрабатывается независимо от других, и какие-либо связи между ними на уровне препроекторной обработки отсутствуют.

Препроцессор последовательно читает порции символов из препроцессорной программы. Если в считанной порции нет препроцессорных операторов и активных препроцессорных переменных или функций, то она передается на выход без изменений. В случае, если в порции имеются активные препроцессорные переменные или функции, они заменяются на соответствующее текущее (для переменных) или вычисляемое (для функций) значение препроцессорной величины. Встречающиеся в порции препроцессорные операторы (отмеченные знаком процента) выполняются в заданной последовательности (с учетом операторов, изменяющих последовательное выполнение). В случае наличия в препроцессорных операторах препроцессорного текста, он обрабатывается, как указано выше.

Выходом препроцессора является, обычно, модуль программы на каком-либо из алгоритмических языков, который направляется на трансляцию или предварительно выдается на внешние носители.

Особенности обработки. Следует обратить внимание на следующие особенности работы препроцессора при выполнении замен в генерируемом тексте.

Повторный просмотр. В коде осуществления замены препроцессорной переменной или функции на их текущее значение, это заменяющее значение вновь подвергается просмотру и обработке содержащихся в нем препроцессорных переменных и функций; новые препроцессорные операторы не должны появляться в программе в результате выполнения замен.

П р и м е р.

```
% DECLARE S CHAR, R FIXED,
      Z ENTRY (FIXED) RETURNS (FIXED);
% Z: PROCEDURE (X) RETURNS (FIXED);
      DECLARE X FIXED;
      RETURN (X*10);
% END Z;
% S = '(U + Z(R))';
% R = 2;
      T = S;
```

При выполнении препроцессорных операторов присваивания (первых выполняемых операторов в данном фрагменте) препроцессорная переменная S получает значение символьной строки, имеющей вид

$(U + Z(R))$,

а R получает значение 2.

После этого во время обработки текста $T = S$; (являющегося оператором PL/1) переменная S заменяется на ее текущее значение,

и должен получиться следующий оператор PL/1:

$$T = (U + Z(R));$$

Но имена Z и R — это имена активных препроцессорных величин (функции и переменной), поэтому препроцессору необходимо произвести дальнейшие замены в обрабатываемом тексте, обратившись при этом к процедуре-функции с аргументом, равным 2. В результате обращения будет возвращено значение $2 \times 10 = 20$, и окончательно получится (без учета добавленных пробелов):

$$T = (U + 20);$$

В ряде случаев повторный просмотр может приводить к тому, что будет осуществляться бесконечная серия замен. Например, для оператора

$$\% S = '(U + S)';$$

в рассмотренном примере сначала получаем:

$$T = (U + S);$$

После повторной замены S имеем:

$$T = (U + (U + S));$$

И далее:

$$T = (U + (U + (U + S)));$$

И так далее. Препроцессором будет зафиксирована ошибка.

Напомним, что замене (первоначальной или повторной) могут подвергаться только препроцессорные переменные и функции.

Если оператор присваивания имеет вид:

$$\% S = '(U + "S")';$$

то в результате замены получается:

$$T = (U + 'S');$$

и дальнейшая замена не производится, поскольку S является не идентификатором, а символом строки. Кроме того, отметим, что если оператор PL/1 имеет вид

$$T = 'S';$$

то замена S в нем вообще не должна производиться.

Таким образом, следует помнить, что в зависимости от того, встречаются ли строки в препроцессорном операторе или вне его, их содержимое обрабатывается препроцессором по-разному. Например, в сочетании с последующим обрабатываемым текстом

$$T = S;$$

из трех операторов:

```
% S='S';  
% S=S+1;  
S='S';
```

первый является недопустимым, а второй и третий — вполне законными на стадии препроцессорной обработки.

В завершение этой темы приведем без особых объяснений пример, демонстрирующий использование эффекта повторного просмотра для построения квазирекурсивных препроцессорных процедур (рекурсивные препроцессорные процедуры не допускаются).

```
% FACTORIAL: PROCEDURE (N) RETURNS (CHAR);  
    DECLARE N FIXED;  
    IF N=0 | N=1 THEN RETURN('1');  
    ELSE RETURN (N||'*FACTORIAL('||N-1||')');  
    % END FACTORIAL;  
% DECLARE FACTORIAL ENTRY (FIXED)  
    RETURNS (CHAR);  
% DECLARE S FIXED;  
% S=7;  
FN=FACTORIAL (S);
```

В результате выполнения данного фрагмента препроцессорной программы на трансляцию будет передан следующий текст:

```
FN=7*6*5*4*3*2*1;
```

Заметим, что возвращаемое из процедуры после первого просмотра, символьное значение имеет такой вид:

```
7*FACTORIAL (6)
```

Для того чтобы сократить количество пробелов, помещаемых препроцессором в текст, можно (в предположении, что $N \leq 99$) вместо первого N в операторе RETURN задать

```
SUBSTR (N, 7, 2)
```

З а м е ч а н и е. Подвергаемый повторному просмотру текст (так же, как и исходный текст) должен отвечать синтаксису препроцессорных средств. Например, нужно следить за тем, чтобы апострофы в тексте составляли пары (см. П. 3, переменные).

Русские буквы. Русские буквы не принадлежат к символам препроцессорных средств, и поэтому они могут встречаться лишь в комментариях и, в некоторых случаях, в строках.

К комментариям относятся только конструкции, ограниченные сочетаниями /* и */ (см. П. 3). Поэтому при использовании препро-

цессорных средств в фортране и алголе, их комментарии, содержащие русский текст, должны быть заключены в пару /* и */ и не должны содержать внутри себя сочетания */. Например:

```
C /* ВЫЧИСЛЕНИЕ ИНТЕГРАЛА */  
  'COMMENT' /* ВЫБОРКА СЛОВА */;  
  'END' /* ПОИСК ТОЧКИ */;
```

Кроме того, нужно предупредить, что буква C, используемая в качестве признака комментария фортрана, может быть принята препроцессором за препроцессорную переменную, и произведена ее замена.

К строкам препроцессор относит только конструкции, ограниченные апострофами (см. П. 3), а с учетом повторного просмотра, в некоторых случаях (см. выше "S") — двойными апострофами ("). Поэтому в фортране приходится отказаться от использования русских букв в формате H и применять в этом случае только строки, заключенные в апострофы. Например, вместо

```
FORMAT (11H_ _УЧЕНИКОВ_ , I13)
```

необходимо использовать

```
FORMAT ('_ _УЧЕНИКОВ_ ', I13)
```

В алголе ЕС кавычки строк изображаются комбинациями '(' и ')', и поэтому внутри строк может находиться русский текст, только заключенный в дополнительные апострофы. Например, вместо

```
OUTSTRING(0,(' _УЧЕНИКОВ_ '))
```

придется задать

```
OUTSTRING(0,(' _УЧЕНИКОВ_ '))
```

что приводит к выдаче слова '_УЧЕНИКОВ_' вместо _УЧЕНИКОВ_, или осуществить выдачу русского текста, используя только процедуру OUTSYMBOL.

Если препроцессор встречается в обрабатываемом (или повторно обрабатываемом) тексте вне комментариев и строк русскую букву, то он фиксирует ошибку, заменяет букву на пробел и продолжает анализ далее.

З а м е ч а н и е. Знак процента может встречаться в комментариях или строках, не вызывая никаких действий.

Строчки выходного текста. Если в результате генерации получается строчка текста процессорной программы, которая не умещается в 71 символ, то остаток образует новую строчку в выходном тексте,

Задание. Задание для PL/1 составляется обычным образом, только среди параметров транслятора должен содержаться параметр MACRO. Например:

```
//PRIM EXEC PL1LFCG,PARM.PL1L='MACRO,LOAD,NODECK'
```

Для фортрана и алгола задание должно состоять из трех шагов, в первом из которых производится макрогенерация с помощью транслятора PL/1 и запись сгенерированной программы на внешний носитель. Для этого должны быть заданы параметры MACRO (включение препроцессора), MACDCK (вывод процессорной программы) и NOCOMP (блокировка трансляции), а также директива DD с именем SYSPUNCH, содержащая характеристики первого вспомогательного набора данных на внешнем носителе. Обычно в качестве вспомогательных наборов данных используются временные последовательные наборы данных.

На втором шаге производится некоторое преобразование полученного набора данных и запись преобразованного текста во второй вспомогательный набор данных. Преобразование необходимо в силу того, что операторы или комментарии фортрана и алгола могут располагаться в тексте, начиная с первой позиции строки, а не со второй, как в PL/1. Поскольку препроцессор PL/1 всегда помещает обработанный текст, начиная со 2-й позиции, необходимо сдвинуть весь текст, помещенный в 1-й вспомогательный набор, на одну позицию влево и записать во 2-й вспомогательный набор. Если в алголе не использовать первую колонку строки для записи программы, то второй шаг можно пропустить (в фортране это невозможно, так как пришлось бы отказаться от комментариев). В любом случае предполагается, что текст программы содержит не более чем по 71 символу в строке.

На третьем шаге производится трансляция (и, может быть, выполнение) программы на фортране или алголе, записанной во втором вспомогательном наборе.

Пример задания для фортрана (для алгола необходимо изменить имена FORT на ALGOL и FORTG на ALGOF).

```
//имя JOB...
//имя1 EXEC PL1LFC,PARM.PL1L='MACRO,NOCOMP,
// MACDCK,SORMGIN=(1,72)'
//PL1L.SYSPUNCH DD характеристики-1-го-набора
//PL1L.SYSIN DD *
//                                     препроцессорная-программа
/*
//имя2 EXEC PGM=имя-программы-сдвига
```

```
//MACIN      DD *,имя1.PL1L,SYSPUNCH
//MACOUT     DD характеристики-2-го-набора данных
//имя3       EXEC FORTGCLG, ...
//FORT.SYSIN DD *,имя2,MACOUT
//GO.SYSIN   DD *
```

исходные-значения-для-программы-на-фортране

```
/*
//
```

Характеристики вспомогательных наборов данных могут иметь, например, следующий вид:

```
DSNAME=&ND1,UNIT=SYSDA,SPACE=(800,(200,20)),
DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),DISP=...
```

При желании все три шага могут быть оформлены в виде каталогизированной процедуры, тогда для обработки и трансляции программы, написанной на фортране или алголе, потребуется только обратиться к этой процедуре. Например, задание может иметь вид:

```
//имя      JOB...
//имя1     EXEC PREPROC
//PL1L.IN DD *
```

препроцессорная-программа

```
/*
//GO.SYSIN DD *
```

исходные-значения-для-процессорной-программы

```
/*
//
```

П.8. Синтаксические формы

Здесь приводятся синтаксические формы препроцессорных средств, позволяющие в сжатой и конкретной форме отобразить типы элементов и порядок их расположения в препроцессорных конструкциях. Формы могут быть использованы как на стадии изучения препроцессорных понятий для проверки и закрепления полученных знаний, так и при практической работе для предупреждения или обнаружения синтаксических ошибок в составленной препроцессорной программе.

Вначале перечисляются используемые обозначения.

Обозначения. Обозначения разделяются на конструктивные и обозначения понятий.

Конструктивные обозначения. Большинство из этих обозначений являются стандартными для документации ЕС ЭВМ.

В приводимых ниже примерах x , y и z представляют синтаксические понятия или литералы (символы и допустимые в рассматриваемых конструкциях сочетания символов).

~ — «соответствует», «это»;

— «или»;

{ } —выбор (альтернатива). Например:

$$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix} \sim \{x | y | z\} \sim x \text{ или } y \text{ или } z$$

$$x\{y|z\} \sim \{xy|xz\}$$

II — необязательный выбор или необязательный элемент конструкции («опция»). Например:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \sim [x|y|z] \sim x \text{ или } y \text{ или } z \text{ или пусто}$$

$$x[y|z] \sim \{xy|xz|x\}$$

() — «полуобязательный» выбор: хотя бы один из элементов в конструкции, заключенный в такие скобки, является обязательным. Например:

$$(x)y(z) \sim \{xyz \mid xy \mid yz\}$$

Сравните:

$$[x]y[z] \sim \{xyz \mid xy \mid yz \mid y\}$$

< > — комментарий, не являющийся частью приводимой конструкции. Например:

$$x \langle y \rangle^2 \sim x^2$$

... — повторение предшествующего элемента. Например:

$$z \dots \sim \{z \mid zz \mid zzz \mid zzzz \text{ (и так далее)}\}$$

$$x\{yz\} \dots \sim \{xyz \mid x y z y z \mid x y z y z y z \text{ (и т. п.)}\}$$

$$\begin{Bmatrix} y \\ z \end{Bmatrix} \dots \sim \left\{ \begin{Bmatrix} y \\ z \end{Bmatrix} \middle| \begin{Bmatrix} y \\ z \end{Bmatrix} \begin{Bmatrix} y \\ z \end{Bmatrix} \middle| \begin{Bmatrix} y \\ z \end{Bmatrix} \begin{Bmatrix} y \\ z \end{Bmatrix} \begin{Bmatrix} y \\ z \end{Bmatrix} \right\} \langle \text{и т. п.} \rangle$$

Повторение является старшей операцией по отношению к выбору ($\{ \}$ или $[\]$ или $()$).

.. — список, то есть перемежающееся повторение двух предшествующих элементов. Например:

$$uz.. \sim \{u | yzu | yzyzu \text{ (и т. п.)}\}$$

$$y, \dots \sim \{y | y, y | y, y, y | y, y, y, y \langle \text{и т. п.} \rangle\}$$

Обозначение понятий. Каждое понятие представляется одним словом или группой слов, соединенных дефисом. Например:

буква, оператор-цикла, имя-переменной

Будут использоваться следующие сокращения для обозначения типов констант, переменных, функций, выражений:

A — арифметический,

Б — битовый (булевский, логический).

С — символьный (литеральный).

Например:

выражение-А ~ выражение-арифметическое

переменная-С ~ переменная-символьная

Формы.

Символы.

буква ~ {А | В | С | D | E | F | G | H | I | J |

K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | □ | @ | #}

цифра ~ {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}

спецзнак ~ {+ | - | * | / | < | = | > | & | ! | ' | , |

: | ; | (|) | ' | - | _ | % | . }

символ ~ {буква | цифра | спецзнак}

символ-ЭВМ ~ {символ | Б | Г | Д | Ж | З | И | Й | Л |

П | У | Ф | Ц | Ч | Ш | Щ | Ъ | Ы | Э | Ю | Я }

Основные элементы.

константа ~ {константа-А | константа-Б | константа-С}

константа-А ~ число ~ [±] цифра ... <максимум 5 цифр>

константа-Б ~ '{0 | 1}' В

константа-С ~ строка ~ ' [символ-ЭВМ <кроме апострофа> | "] ... '

идентификатор ~ имя ~ буква [буква | цифра | _] ...

<максимум 31 символ>

имя ~ {имя-переменной | имя-входа}

метка ~ идентификатор-меточный <идентификатор>

переменная ~ имя-переменной

переменная ~ {переменная-А | переменная-С}

функция ~ {имя-входа [(аргумент, ...)]

| SUBSTR(выражение-С, выражение-А
, выражение-А)}

функция ~ {функция-А | функция-С}

аргумент ~ {константа-А | константа-С | переменная}

комментарии ~ /* символ-ЭВМ... <кроме сочетания */ */

<ставятся вне констант и имен>

Выражения.

выражение ~ {выражение-А | выражение-Б | выражение-С}

выражение-А ~ {константа-А | переменная-А | функция-А
| (выражение-А) | [±] выражение-А

| выражение-А {+ | - | * | /} выражение-А}

выражение-Б ~ {константа-Б

| (выражение-Б) | \neg выражение-Б
| выражение-А \leq выражение-А
| выражение-С \leq выражение-С
| выражение-Б {& | !} выражение-Б}
 $\langle \leq \sim \{ < | < = | > | > = | \neg = \} \rangle$

выражение-С ~ {константа-С | переменная-С | функция-С

| (выражение-С)
| выражение-С !! выражение-С}

⟨старшинство-операций ~ { \neg || * | / || + | - || || \leq || & || }⟩

Операторы.

оператор-активации ~

~ % [метка:]... ACTIVATE{имя-переменной |
имя-процедуры},...;

оператор-деактивизации ~

~ % [метка:]... DEACTIVATE{имя-переменной |
имя-процедуры}, ...;

оператор-присваивания ~

~ % [метка:]... переменная = {выражение-А | выражение-С};

оператор-перехода ~ % [метка:] ... {GOTO} метка;
GOTO

оператор-пустой ~ % [метка:]...;

оператор-условный ~

~ % [метка:] ... IF выражение-Б % THEN оператор
[% ELSE оператор]

оператор-составной ~

~ % [метка:] ... DO; {оператор | текст} ...
% END [метка-для-DO];

оператор-цикла ~

~ % [метка:] ... DO переменная-А = спецификация;

{оператор | текст} ...

% END [метка-для-DO];

спецификация ~ выражение-А [BY выражение-А] TO

выражение-А

оператор-включения ~

~ % [метка:] ... INCLUDE { имя-DD (имя-раздела) } ...
имя-раздела

оператор-вызова ~ функция ~

~ {имя-процедуры [(аргумент, ...)]

| SUBSTR(аргумент, выражение-А, выражение-А)}

аргумент ~ текст <скобки — парами; запятые — в скобках>

Описания (операторы описаний).

описание-имени ~ % { DECLARE } { {имя | (имя; ..)} описатель}, ..
DCL
описатель ~ {описатель-переменной | описатель-входа}
описатель-переменной ~ [FIXED | CHAR[ACTER]]
описатель-входа ~ ENTRY[(описатель-переменной, ..)]
RETURNS(описатель-выхода)
описатель-выхода ~ {FIXED | CHAR[ACTER]}
описание-процедуры ~ процедура ~
~ % {имя-входа:} ... PROC[EDURE] [(параметр, ..)]
RETURNS(описатель-выхода);
{[описание-параметров <без %>] | оператор <без %>}} ...
<кроме CTIVATE, DEACTIVATE, INCLUDE>
<оператор-возврата ~ RETURN({выражение-A |
выражение-C});>
% END [имя-входа];
параметр ~ переменная

Программа и текст.

программа ~ {(текст) (описание | оператор) ... (текст)} ...
текст ~ {(символ <без'> ...) ('[символ-ЭВМ <без'> |" ...')
(символ <без'> ...) } ...

П.9. Сравнение препроцессорных и процессорных средств PL/1

Для тех читателей, которые знакомы с PL/1, ниже перечисляются ограничения и особенности, имеющиеся в препроцессорных средствах, по сравнению со средствами языка PL/1 для ОС ЕС ЭВМ (процессорные средства).

Символы. Среди основных символов, используемых в конструкциях препроцессора, отсутствует точка.

Данные и константы. Используются лишь арифметические и строчные данные. Арифметические константы имеют вид действительных десятичных целых чисел, в которых не может быть более 5 цифр. В битово-строчных и символично-строчных константах отсутствует повторитель строки.

Идентификаторы и метки. «Массивные» метки, имеющие вид переменной с индексами, отсутствуют.

Переменные. Используются только простые переменные. Переменные с индексами, а также переменные над массивами и структурами отсутствуют. Разрешенные типы переменных: арифметические (см. константы) и символично-строчные; битово-строчные отсутствуют.

Функции. Имеется только одна встроенная функция — SUBSTR, при обращении к которой всегда должны задаваться все 3 аргумента. Аргументами функций, введенных программистом, могут быть только арифметические и символично-строочные константы и переменные; возвращаемое значение — арифметическое или символично-строочное.

Комментарии. Те комментарии, которые расположены в препроцессорных операторах, в процессорный текст не попадают.

Выражения. Операция возведения в степень отсутствует; деление всегда дает целый результат (с отброшенной дробной частью). Наряду с арифметическими и строочными операциями допускаются и логические операции (например, над результатами операций сравнения). Могут автоматически включаться преобразования типа.

Операторы. Все препроцессорные операторы (исключение — вызов процедуры) начинаются со знака процента (%), вслед за которым может следовать несколько меток.

Среди выполняемых операторов используются только операторы присваивания, перехода, вызова процедуры, пустой, условный, групповой.

Список левой части в операторе присваивания запрещен. Знак % ставится в условном операторе и перед ключевыми словами THEN и ELSE, а в групповом операторе — и перед END.

В групповом операторе циклического типа не допускается конструкция с WHILE; список цикла (несколько спецификаций цикла, разделенных запятыми) не допускается. Среди операторов, заключенных в операторные скобки %DO; и %END;; может находиться и обрабатываемый текст. (Синтаксические формы для группового оператора (составного и цикла), приведенные в П.8, не отражают всех возможностей этого оператора.)

Оператор вызова процедуры пишется без знака процента и без ключевого слова CALL и по своему действию совпадает с функцией, возвращающей в место ее вызова символично-строочное значение; особенности задания аргументов см. в П.5.

Все другие процессорные операторы языка PL/1 в препроцессоре отсутствуют.

Введены новые операторы: активации, дезактивации (П.7), включения текста (П.6).

Описания. В операторах описания атрибуты точности и длины не указываются. Для целых величин задается только атрибут FIXED, точность всегда предполагается равной 5. Символьные величины всегда имеют изменяемую длину, максимум которой устанавливается при реализации транслятора (атрибут VARYING не задается). Никакие правила умолчания не действуют.

В описании входа, которое является обязательным, атрибуты ENTRY и RETURNS задаются всегда.

При описании процедуры (функции) все операторы, составляющие тело процедуры, пишутся без знака процента; опция RETURNS является обязательной. Вложенные и внешние процедуры запрещаются.

Препроцессорная программа имеет вид последовательности препроцессорных операторов, описаний и обрабатываемого текста; заголовков головной (MAIN) процедуры (PROCEDURE-оператор) и оператор конца (END-оператор), имеющиеся в процессорных операторах PL/1, не пишутся. Описания переменных и процедур должны встречаться до того, как в препроцессорной программе используется эта величина.

В заключение приведем для справок используемые в препроцессорных средствах ключевые слова (табл. 2), которые кратко отображают возможности, имеющиеся в препроцессоре транслятора с языка PL/1 ОС ЕС.

Т а б л и ц а 2

Ключевые слова препроцессорных средств PL/1

Ключевое слово	Сокраще- ние	Использование	Ссылка
ACTIVATE	CHAR	оператор (новый)	П. 4
BY		см. оператор DO	П. 4
CHARACTER	DCL	атрибут	П. 3
DEACTIVATE		оператор (новый)	П. 4
DECLARE		оператор описания	П. 3, П. 5
DO		оператор	П. 4
ELSE		см. оператор IF	П. 4
END		см. операторы DO и PROCEDURE	П. 4, П. 5
ENTRY		атрибут	П. 5
FIXED		атрибут	П. 3
GO TO	GOTO	оператор	П. 4
IF		оператор	П. 4
INCLUDE	PROC	оператор (новый)	П. 6
PROCEDURE		оператор	П. 5
RETURN		оператор	П. 5
RETURNS		атрибут	П. 5
THEN		см. оператор IF	П. 4
TO		см. оператор DO	П. 4

ЛИТЕРАТУРА

1. Брукс Э. Как проектируются и создаются программные комплексы.— М.: Наука, 1979.
2. Средства отладки больших систем / Под ред. Р. Растина.— М.: Статистика, 1977.
3. Ершов А. П. Технология разработки систем программирования.— В кн: Системное и теоретическое программирование, Новосибирск: СО АН СССР, 1972.
4. Дал У., Дейкстра Э., Хоор К. Структурное программирование.— М.: Мир, 1975.
5. Вирт Н. Систематическое программирование.— М.: Мир, 1977.
6. Иодан Э. Структурное проектирование и конструирование программ.— М.: Мир, 1979.
7. Хьюз Дж., Мичтом Дж. Структурный подход к программированию.— М.: Мир, 1980.
8. Майерс Г. Надежность программного обеспечения.— М.: Мир, 1980.
9. Brown A. R., Sampson W. A. Program debugging.— New York: Mac Donald, 1973.
10. Средства отладки в ОС ЕС ЭВМ.— М.: Статистика, 1979.
11. Гребенников Л. К., Лебедев В. Н. Решение задач на ПЛ/1 в ОС ЕС,— М.: Финансы и статистика, 1981.
12. Безбородов Ю. М. Сравнительный курс языка PL/1.— М.: Наука, 1980.
13. Брич З. С., Капилевич Д. В., Котик С. О., Цагельский В. И. Фортран ЕС ЭВМ.— М.: Статистика, 1978.
14. Митрофанов В. В., Одинцов Б. В. Программы обслуживания ОС ЕС ЭВМ.— М.: Статистика, 1977.
15. Халилов А. И., Ющенко А. А. Алгол-60.— Изд. 3-е.— Киев: Вища школа, 1979.
16. Демин В. Ф., Добролюбов Л. В., Степанов В. А. Системы программирования на алголе.— М.: Наука, 1977.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- Алгол 46, 48, 52, 54—56, 60, 85—87, 89, 97, 105, 126, 127, 180
 - , режим TEST 56, 57
 - , — TRACE 56
- Алгоритмизация 10, 11, 16, 18, 23, 103, 116—118, 127, 136, 151
- Ассемблер 47, 81, 82, 100
- Блок 26, 32, 35, 36
- Блок-схема 11, 12, 27, 70, 128, 129
- Имитатор 35, 36, 132, 138, 139
- Инструкция 10, 14, 135, 149
- Исправления 14, 16, 72, 79
 - алгоритма 72
 - , локальность 92, 93
 - модулей 86
 - программы 73
 - — контекстные 74, 77—79
 - — построчные 73—76
- Кодирование 12
 - нисходящее 131
- Комментарии 83, 130
- Контроль алгоритма 12
 - индексов 57
 - программы 14, 16, 22, 30
- Локализация ошибок 14, 16, 39, 66, 70, 130, 131
- Модернизация 15, 16
- Модуль 86, 87, 95, 96
- Модульность 86, 87, 89, 127
- Оператор перехода (GO TO) 99, 101
- Отладка 9, 13, 16, 20, 141
 - , дневник 70, 71
 - модулей 86
 - , план 18, 20, 138
- Отслеживание 67—69
- Отчет 15, 16
- Оформление программы 14, 16, 20
- Ошибка 12—14, 18, 32
 - , защита 84, 85
 - , место 67
 - препараты 13, 29
 - , природа 69
 - , симптом 66
- Печать отладочная 66
 - — аварийная 40, 41—46, 59, 61, 62
 - — внешней памяти 57
 - — в узлах 40, 48—52, 59, 61
 - — дампа 47
 - —, идентификация 62
 - — исходных данных 69
 - —, минимизация 62
 - — текста 13, 29
 - — финишная 44
- Препарация 12, 16, 20
- Процессор 159
 - средства 46, 47, 50—52, 65, 66, 74, 77—79, 81, 85, 88—93, 95, 99—101, 141, 142, 159—188
- Проверка 23, 70
 - , точность 33, 34
- Программа простая 98
 - самодокументированная 83
 - , строение 88

Программа UPDATE 52, 74—78,
95, 142, 148, 149, 151
Программирование 12, 16, 18, 20
—, стандарты 84
Проектирование 10, 12, 20, 134,
146—149
— нисходящее 103, 127, 129, 130
Прокрутка 40, 55, 59, 61
— арифметическая 55
— логическая 56
— «сухая» 23—28, 143, 144
Просмотр 22
Псевдокод 128

Сверка 13, 63, 64
Слежение 40, 52, 59, 61
— арифметическое 52—54
— логическое 54, 55
Средства исправлений 73, 74
— отладки (локализации 41),
58—60, 65
— — —, включение 65
— — —, классификация 57
— — —, сравнение 61
Структура дополнительная 99
— нестандартная 101
— стандартная 96, 97
Структурированность 96, 127,
137
Счет 15, 16

Тест 30, 37, 38, 70, 143
Тестирование 30, 31, 73
— восходящее 35, 131
— модулей 86
— нисходящее 35, 36, 131, 132,
157, 158
—, планирование 35, 136, 151,
157
—, правила 32—35
Техническое задание (ТЗ) 9, 10,
130, 134, 146
Трансляция 13, 16, 29
— стадии 159

Фортран 60, 85—87, 89, 99—101,
114—116, 139, 140, 141, 160—
162, 175, 180
—, оператор AT 51, 54
—, — CALL DUMP 45
—, — DISPLAY 51
—, — IF 46, 48, 51, 100, 101
—, — NAMELIST 45, 46, 62, 65
—, описатель начальных зна-
чений 41
—, подпрограмма BLOCK DATA
41
—, — DVCHK 44, 45
—, — OVERFL 44, 45
—, режим INIT 53, 56
—, — SUBCHK 57
—, — SUBTRACE 54
—, — TRACE 54, 56
—, — UNIT 64

Язык алгоритмический 128
— вспомогательный 105
— высокого уровня 81
— программирования 11
— промежуточный 11

PL/1 66, 85—87, 89, 91, 94, 97,
99, 100, 105, 111—114, 121—
126, 139—141, 160, 175, 180
—, оператор GET DATA 65
—, — IF 46, 48, 50, 63
—, — ON 42, 46, 49, 50, 63, 101
—, — PUT DATA 42—44, 46,
62
—, — SIGNAL 43
—, описатель INITIAL 41, 93,
125
—, ситуация 43, 85
—, — CHECK 49, 50, 52—54,
56, 63, 142
—, — ERROR 42—44
—, — FINISH 43
—, — SUBSCRIPTRANGE 43,
57

Безбородов Юрий Михайлович
ИНДИВИДУАЛЬНАЯ ОТЛАДКА ПРОГРАММ

Редактор *О. Ю. Меркадер.*

• Технический редактор *С. Я. Шкаяр.*

Корректоры: *Л. И. Назарова, А. Л. Ипатова.*

ИБ № 12081

Сдано в набор 03.09.81. Подписано к печати
25.12.81. Т-30890. Формат 84×108^{3/4}. Бумага
тип. № 2. Литературная гарнитура. Высокая
печать. Условн. печ. л. 10,08. Уч.-изд. л. 11,62.
Тираж 65000 экз. Заказ № 3251. Цена 70 коп.

Издательство «Наука»

Главная редакция

физико-математической литературы

117071, Москва, В-71, Ленинский проспект, 15

Ордена Октябрьской Революции
и ордена Трудового Красного Знамени
Первая Образцовая типография
имени А. А. Жданова Союзполиграфпрома
при Государственном комитете СССР
по делам издательств, полиграфии
и книжной торговли.
Москва, М-54, Валовая, 28
Отпечатано в тип. № 2. Изд-ва «Наука»,
Москва, Шубинский пер., 10. Заказ № 1254

ИЗДАТЕЛЬСТВО «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
117071, Москва, В-71, Ленинский проспект, 15

ГОТОВИТСЯ К ПЕЧАТИ
в 1982 году

Гуляев А. В., Макаров-Земляиский Н. В.,
Машечкин И. В. Диалоговый комплекс программ
Краб/ Под ред. Л. Н. Королева.— М.: Наука, Главная
редакция физико-математической литературы.

Книга посвящена описанию возможностей системы
общего назначения Краб примерительно к ЭВМ типа
БЭСМ-6. Представлен инструктивно-справочный материал
на уровне как пользователей, так и лиц, сопровождаю-
щих систему. Преимущества системы: в малой ресурсо-
емкости, высокой надежности, защищенности информа-
ции, в возможностях сопровождения, в бюджетно-стати-
стической службе, в работе непосредственно с терминала.

Для научных работников, инженеров, программистов.

*Предварительные заказы на эту книгу принимаются
без ограничения магазинами Книготорга и Академкниги.*

70 коп.

5ВГ66
Б-391