

**БИБЛИОТЕЧКА  
ПРОГРАММИСТА**

**В. А. ВАСИЛЬЕВ**

# **Язык Алгол-68**

**основные понятия**





БИБЛИОТЕЧКА  
ПРОГРАММИСТА

---

В. А. ВАСИЛЬЕВ

# ЯЗЫК АЛГОЛ-68

## ОСНОВНЫЕ ПОНЯТИЯ

Под редакцией С. С. ЛАВРОВА



ИЗДАТЕЛЬСТВО «НАУКА»  
ГЛАВНАЯ РЕДАКЦИЯ  
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ  
МОСКВА 1972

**Язык АЛГОЛ-68 (основные понятия), В. А. Васильев.**  
Главная редакция физико-математической литературы изд-ва «Наука», М., 1972.

Книга содержит изложение системы понятий и разбор основных конструкций универсального алгоритмического языка АЛГОЛ-68, отражающего общие современные представления о вычислительных процессах и способах их изображения.

Не являясь учебным пособием, книга дает читателю общее, но достаточно полное представление об АЛГОЛе-68 и вводит его в круг интересных и актуальных вопросов (часто дискуссионного характера), касающихся организации современных языков программирования.

Она будет полезна как профессиональным программистам, так и математикам, инженерам и студентам разных специальностей, интересующимся теорией и практикой программирования.

К читателю не предъявляется никаких специальных требований, кроме элементарного знания АЛГОЛа-60 и наличия некоторой культуры программистского мышления.

*Владимир Алексеевич Васильев*

Язык АЛГОЛ-68. Основные понятия

(Серия «Библиотечка программиста»)

М., 1972 г., 128 стр.

Редактор Г. Я. Пирогова

Техн. редактор К. Ф. Брудно

Корректоры Э. В. Астонеева, Л. С. Сомова

Сдано в набор 8/VI-1972 г. Подписано к печати 22/VIII-1972 г. Бумага 84×108/32

Физ. печ. л. 4 Условн. печ. л. 6,72. Уч.-изд. л. 6,77.

Тираж 55000 экз. Т-14710. Цена книги 43 к.

Заказ № 844

Издательство «Наука»

Главная редакция физико-математической литературы  
117071, Москва, В-71, Ленинский проспект, 15

2-я типография издательства «Наука», Москва, Шубинский пер., 10

## ОГЛАВЛЕНИЕ

Предисловие . . . . .	5
Введение . . . . .	7
<b>Г л а в а I. Значения . . . . .</b>	<b>11</b>
§ 1. Простые значения . . . . .	11
§ 2. Структурные значения . . . . .	13
§ 3. Кратные значения . . . . .	13
§ 4. Имена . . . . .	16
§ 5. Процедурные значения (рутины) .	19
§ 6. Виды значений . . . . .	20
§ 7. Области действия значений . . .	23
<b>Г л а в а II. Элементы описательной техники</b>	<b>26</b>
§ 1. Строгий язык и язык представлений. Синтаксис строгого языка . . . . .	26
§ 2. Метаязык и использование метапонятий для образования синтаксиса строгого языка. Виды	28
§ 3. Действия . . . . .	34
§ 4. Расширения и расширенный язык .	36
<b>Г л а в а III. Основные конструкции . . . . .</b>	<b>38</b>
§ 1. Идентификаторы . . . . .	39
§ 2. Обозначения . . . . .	39
§ 3. Вызовы . . . . .	48
§ 4. Вырезки . . . . .	50
§ 5. Выделения полей . . . . .	58
§ 6. Присваивания . . . . .	61
§ 7. Ядра , , , , , . . . . .	67

§ 8. Описатели . . . . .	68
§ 9. Генераторы . . . . .	76
§ 10. Описания идентичности . . . . .	78
§ 11. Формулы и описания операций и приоритетов . . . . .	87
§ 12. Индиканты вида и описания вида	92
§ 13. Общее строение выражений и описаний . . . . .	96
<b>Г л а в а IV. Приведения . . . . .</b>	<b>103</b>
§ 1. Смысл и общая схема аппарата приведений . . . . .	103
§ 2. Типы приведений . . . . .	110
§ 3. Ограничения приведений . . . . .	117
<b>Г л а в а V. Объединенные виды и отношения согласования . . . . .</b>	<b>120</b>
§ 1. Объединенные виды . . . . .	120
§ 2. Отношения согласования . . . . .	126
<b>Литература . . . . .</b>	<b>128</b>

## ПРЕДИСЛОВИЕ

Настоящая книга не является учебным пособием, призванным помочь читателю овладеть новым алгоритмическим языком и использовать его на практике. Цель ее заключается в том, чтобы дать общее (но достаточно полное) представление об АЛГОЛе-68 на уровне основных идей, отвлекаясь от многочисленных деталей, несущественных с точки зрения понимания духа языка (но, к сожалению, необходимых для его сознательного использования и тем более реализации). Следует заметить, что в процессе работы над языком, начиная от «Предварительного проекта» [1] и до «Сообщения» [2], число измененных в нем деталей весьма велико, в то время как идейный скелет почти не претерпел изменений. Это похоже на процесс отладки программы, когда производятся различные вставки, удаления, сдвиги и т. п., но не затрагиваются основные алгоритмы. Мы увидим, что эта аналогия не так груба, как могло бы показаться, и в дальнейшем еще не раз воспользуемся ею.

Имеется несколько доводов в пользу указанной направленности книги. Во-первых, совокупность идей, лежащих в основе АЛГОЛа-68, очень интересна сама по себе. Во-вторых, непосредственное ознакомление с «Сообщением» [2] требует от читателя больших усилий, так как основную часть текста можно сравнить с большой и хорошо закрученной программой, в которой (как это часто бывает в программах) тесно переплетены существенное и второстепенное. Данная работа может подготовить желающих

к полному изучению языка по официальным документам. Наконец, текст [2] тоже не до конца «отлажен» и еще будет дорабатываться. Поэтому рассмотрение всех его деталей (т. е. написание учебника) в настоящее время преждевременно. В то же время возможные доработки, насколько нам известно, не будут носить принципиального характера, поэтому данная работа может подготовить читателя и к чтению последующих версий языка.

*В. Васильев*

## ВВЕДЕНИЕ

Создание алгоритмических языков преследует разные цели. Некоторые предназначены для эффективного описания узкого класса алгоритмов, авторы других, не претендуя на глубину идей и логическую цельность, стремятся к максимальной простоте использования и реализации и т. п. Целесообразность разработки подобных языков и сами эти языки — личное дело их авторов, которые обычно это хорошо понимают и, информируя окружающих о своем детище, говорят примерно следующее: «Мы используем этот язык для того-то и того-то, нам это удобно, если для ваших целей это тоже удобно, то очень хорошо, а если нет, то используйте что-нибудь другое или придумайте свое» \*).

АЛГОЛ-68 был задуман как универсальный язык программирования, «обслуживающий многих людей во многих странах» [2], и официальный преемник АЛГОЛа-60. Он разрабатывался под эгидой Международной федерации по обработке информации (ИФИП). Язык такого масштаба, разумеется, уже не личное дело авторов, он должен отражать общие современные представления о вычислительных процессах и способах их изображения и использовать подходящую для этого *описательную технику*, т. е. методы задания синтаксиса и семантики. Разработка такого языка, базирующегося на новой системе понятий и новой описательной технике (а именно таков АЛГОЛ-68), — дело чрезвычайно сложное и длительное, требую-

---

\*) Типичным примером такого языка может служить ЭПСИЛОН, разработанный в Вычислительном центре Сибирского отделения АН СССР (см. И. В. Поттосин, А. Ф. Рар, В. Л. Катков «ЭПСИЛОН — система автоматизации программирования задач символьной обработки», Труды 1-й Всесоюзной конференции по программированию, Киев, 1968).

щее многочисленных дискуссий, в которых должны принимать участие тоже «многие люди из многих стран» (что и имело место при разработке АЛГОЛа-68). Поэтому естественно задать по крайней мере два вопроса.

Во-первых, зачем вообще понадобилось создавать новый универсальный язык, качественно отличный от уже общепринятого АЛГОЛа-60, почему было не пойти по пути расширения последнего, вводя операции над новыми объектами, но оставляя неизменной общую организацию языка? Другими словами, чем принципиально неудовлетворителен АЛГОЛ-60 с современной точки зрения?

Во-вторых, каким мы в общих чертах хотим видеть новый язык и насколько отвечает нашему желанию АЛГОЛ-68?

Сейчас, отвечая на эти вопросы, остановимся только на следующих двух пунктах.

**1. Виды и значения.** В АЛГОЛе-60 с идентификаторами связываются объекты некоторых *видов*, как-то: целая переменная, логический двумерный массив, процедура с одним вещественным параметром и т. д. \*). С переменными связываются другие объекты — их *значения*. Значения могут меняться в рамках, определяемых видом, и манипуляции со значениями переменных составляют основную часть работы с ними. С процедурами же, например, не связывается никаких значений, и поэтому работа с процедурами сводится в конечном счете к схеме «описать-выполнить». (По существу это эквивалентно тому, как если бы процедурные значения имелись, и с идентификатором процедуры связывалось бы одно такое значение, без возможности варьирования.) А что, если бы и у процедур было свое множество значений, и эти значения можно было бы присваивать идентификаторам процедур подобно тому, как идентификатору переменной присваивается в качестве значения число 1 \*\*)? Это, во-первых, увеличило бы гибкость языка, а, во-вторых, сделало бы его более единообразным, устранив неравноправие пере-

---

\*) В АЛГОЛе-60 термина «вид» нет, под видом, грубо говоря, понимается совокупность класса и типа данного объекта и видов объектов, с ним связанных (например, параметров процедуры).

\*\*) Зачатки такого подхода в АЛГОЛе-60 есть. Например, замена формального параметра, не включенного в список значений фактическим параметром во время выполнения оператора процедуры по существу эквивалентна присваиванию этому формальному параметру в качестве значения того текста, который представлен фактическим параметром. (Прим. ред.)

менных и процедур. Проводя эту мысль до конца, спросим себя: а нельзя ли устроить язык так, чтобы в нем любому виду соответствовало определенное множество значений? Этим мы, во-первых, добились бы максимального единообразия, так как один аппарат общих манипуляций со значениями (например, механизм присваиваний) обслуживал бы объекты всех видов, а, во-вторых, обеспечили бы себе одинаковую гибкость использования идентификаторов разных видов в смысле возможности изменения их значений. И то, и другое, очевидно, весьма заманчиво.

Можно ли без особых хлопот так модернизировать АЛГОЛ-60?

В АЛГОЛе-60 есть арифметические и логические выражения, но нет, скажем, выражений «процедурных», так как у процедур нет значений. Но если каждому виду соответствуют значения, то естественно иметь в языке и выражения любого вида (т. е. имеющие значения этого вида), в частности, вида процедур или массивов (например,  $A \times B$ , где  $A$  и  $B$  — идентификаторы матриц, а  $\times$  — знак умножения матриц). А тогда естественно разрешить таким выражениям занимать позиции, ранее занимавшиеся только идентификаторами процедур или массивов, подобно тому, как арифметическое выражение может занимать, например, позицию вещественного фактического параметра. Тогда переменная с индексами сможет иметь, например, такое строение:

$$(A \times B) [i, j].$$

Кроме того, ввиду равноправия видов (в смысле наличия значений) становится естественной возможность формировать массивы из компонент любого вида, причем желателен единый аппарат для работы с массивами разных видов. Наконец, становится ненужным имеющееся в АЛГОЛе-60 принципиальное различие описаний типа, массивов и процедур.

Действительно, можно говорить просто об описании объекта того или иного вида, с которым либо не связывается сначала никакое конкретное значение (как в описании типа), либо такое начальное значение дается (по существу как в описании процедуры). Более того, можно считать, что начальное значение дается всегда, только в первом случае оно будет неопределенным.

(Что касается описания переключателя в АЛГОЛе-60, то оно при таком подходе оказывается излишним и, как

мы далее увидим, может быть заменено описанием массива процедур.)

Мы видим, что пытаться реализовать такую гибкую организацию языка в узких рамках АЛГОЛа-60 нецелесообразно. А АЛГОЛ-68 именно так и задуман, и эти идеи в нем последовательно выдержаны.

**2. Строгость семантического определения.** По этому поводу АЛГОЛ-60 подвергался многочисленной и справедливой критике. Действительно, в то время как формализация синтаксиса АЛГОЛа-60 явилась его неоценимым достоинством, неформальные словесные предписания его семантики, местами нечеткие и бьющие на очевидность, содержат много подводных камней (см. работу [3]), породивших различные трактовки отдельных понятий даже среди самих авторов. Своей жизнеспособностью АЛГОЛ-60 обязан тому, что в основном в семантике речь идет о вещах, хорошо всем знакомым из практики вычислений (именно там, где появлялось что-то новое, как, например, побочный эффект, и начинались неприятности).

Но если мы организуем язык, как описано выше, то достигнутые единообразие и гибкость неизбежно вызовут к жизни множество новых возможностей, эффект использования которых мы можем заранее и не предвидеть, но тем не менее обязаны шаг за шагом формально определить происходящее.

В АЛГОЛе-68 это тоже последовательно выдержано. Для каждого синтаксически определяемого класса конструкций в разделе «Семантика» сугубо алгоритмическим путем задается *выполнение (elaboration)* представителей этого класса. Несмотря на то, что читать такую семантику без пояснений довольно трудно (как и разбираться в чужой программе), ею достигается важная цель — полнота и недвусмысленность описания. Пояснения же, конечно, давать не возбраняется, и тогда чтение облегчается, но не в ущерб строгости.

Рассмотренные два пункта представляются нам наиболее глобальными из тех, которые привели в конце концов к созданию АЛГОЛа-68. В дальнейшем мы еще встретимся с другими, более локальными концепциями, лежащими в основе его организации и отражающими отдельные аспекты современной идеологии вычислений. Этим постепенно будут наращиваться ответы на поставленные выше вопросы.

## ГЛАВА I

### ЗНАЧЕНИЯ

При описании процесса выполнения каких-либо конструкций часто говорится, что некоторые их составные части *обладают значениями*. Термин *обладать* — первичный, он означает только, что установлена некоторая связь между частью программы и объектом, называемым *значением*. Основные действия, составляющие выполнение, — это установление или устранение такой связи и манипуляции со значениями, которые таким образом образуют внутреннюю, семантическую «кухню» языка. (Значения иначе еще называются *внутренними* объектами, в отличие от *внешних* — частей программы.)

Поэтому прежде чем рассматривать сами языковые конструкции, целесообразно рассмотреть, какие различаются значения, и что они собой представляют (т. е. с чем работает язык посредством своих конструкций).

Значения делятся на следующие шесть классов: *простые, структурные, кратные, процедурные (или рутинные), форматы и, наконец, имена*.

Значения, называемые форматами, используются только для целей ввода-вывода, и мы на них останавливаться не будем. Не будем также в дальнейшем рассматривать и конструкции, связанные с использованием форматов.

Остальные пять классов значений рассматриваются ниже.

#### § 1. Простые значения

Простые значения делятся на *арифметические* — целые и вещественные числа, *логические* — слова *true* (*истина*) и *false* (*ложь*) и *литерные* — некоторые знаки, в частности, буквы и цифры.

Особенностью арифметических значений по сравнению с АЛГОЛом-60 является то, что им приписывается некоторая *длина* (положительное целое число). Таким

образом, арифметическое значение можно рассматривать как пару (*число, длина*). Роль длин состоит в следующем. Для каждой длины считается заданным свой *диапазон*, т. е. наибольшее целое и вещественное число, могущее иметь эту длину (конкретные диапазоны фиксируются реализацией). В языке могут использоваться только такие арифметические значения, в которых число не превосходит диапазона его длины. Далее, для каждой длины считается заданным наименьшее положительное вещественное число (тоже фиксируемое реализацией), еще «отличимое» от нуля (точнее, такое, что результат сложения его с единицей той же длины еще будет больше единицы, а результат вычитания из единицы — меньше).

Ясно, что все это связано со способом и точностью представлений чисел различных длин в конкретной реализации. (АЛГОЛ-60 почти полностью игнорировал тот факт, что программы в конечном счете выполняются на машинах, поэтому аналогичные понятия там отсутствуют.)

Над числами одинаковой длины считаются определенными «в смысле численного анализа» обычные арифметические операции, дающие в результате число той же длины (в случае выхода за диапазон такое значение не может быть использовано и объявляется неопределенным). Определенными считаются также и операции отношений. Заметим, что сложение не считается определенным *a' priori*, так как оно определяется через вычитание. Точно так же умножение целых чисел не считается данным, ибо определяется через сложение. Поэтому для целых чисел из арифметических операций считается определенным только вычитание (деление определено только для вещественных чисел), а для вещественных — вычитание, умножение и деление. Из операций отношений первичной считается только  $<$ , остальные определяются через нее.

Так же, как и в АЛГОЛе-60, для каждого целого числа существует *эквивалентное* (т. е. равное ему с точностью до погрешностей представления) вещественное число той же длины, а результат арифметической операции считается целым только тогда, когда оба операнда целые.

Арифметические операции над числами разной длины не определяются. Однако числу меньшей длины с помощью специальной операции может быть приписана большая длина (что в реализации может повлечь изменение способа представления), после чего арифметическая операция

окажется возможной. В некоторых случаях, кроме операции «удлинения», определена и обратная операция «укорочения». Поэтому и не определяется, как производятся арифметические операции с числами разных длин (по наибольшей или по наименьшей длине), а требуется предварительное явное выравнивание длин.

## § 2. Структурные значения

*Структурное значение* — это упорядоченная совокупность других значений, рассматриваемая как единое целое. Компоненты структурного значения называются его *полями*. Полем может быть произвольное значение, в частности, снова структурное. Например, можно рассмотреть структурное значение с двумя полями, первое из которых — целое, а второе — структурное значение с двумя вещественными полями. Заметим, что у структурного значения может быть только одно поле (и такая возможность часто оказывается удобной)\*).

Наличие структурных значений в совокупности с именами (см. § 4), могущими, как и любые другие значения, быть полями, дает, в частности, очень интересную и мощную возможность организовать работу с так называемыми *списочными структурами* (или *списками*), причем со сколь угодно сложными. С помощью таких списков могут быть изображены действия над весьма разнообразными объектами.

В АЛГОЛе-60 аналог структурного значения отсутствует.

## § 3. Кратные значения

В АЛГОЛе-60 массивом называется «упорядоченное множество чисел или логических значений». Ничего больше в это понятие на уровне внутреннего объекта (т. е. значения) не входит, не уточняется даже характер упоря-

---

\*) Типичным примером структурного значения может служить комплексное число. Полями этого значения являются два вещественных числа — его вещественная и мнимая части. Известно, что при работе с комплексными величинами удобны обозначения:  $\operatorname{Re} z$  и  $\operatorname{Im} z$  для компонент комплексного числа  $z$ .

По той же причине в АЛГОЛе-68 с каждым структурным видом связываются наименования (выделители) его полей, позволяющие строить удобные обозначения для любой компоненты структурного значения данного вида. (*Прим. ред.*)

дочения. Понятие размерности появляется уже на уровне внешнего объекта — идентификатора массива (размерность определяется как число граничных пар в описании массива). Такая логическая незавершенность не создает особой беды в АЛГОЛе-60, так как средства работы с массивами там очень бедны и сводятся в конечном счете к работе с их отдельными элементами. В АЛГОЛе-68 предусмотрен более гибкий аппарат, позволяющий производить такие преобразования, как получение подмассива, конструирование нового массива из произвольных значений одного и того же вида и т. п. Поэтому необходимо определить, что будут означать эти преобразования с формальной точки зрения, и уточнить для этих целей прежде всего само понятие массива. Так как в АЛГОЛе-68 проводится линия четкого разграничения внутренних объектов (значений), ответственных за семантическую интерпретацию, и внешних объектов (частей программы), которым разрешено только обладать значениями, то такое уточнение проводится на уровне значений. А именно, определяются особые *кратные* значения, в которых содержится вся нужная информация.

*Кратное значение* состоит из некоторого числа других значений, называемых его *элементами*, и специальной управляющей части, называемой *паспортом*.

Каждому элементу поставлено в соответствие некоторое целое число, называемое *номером* элемента. Другими словами, элементная часть кратного значения — это совокупность пар (*элемент, номер* \*). Элементами могут быть произвольные не кратные значения (в кратных элементах нет необходимости в силу наличия обычной возможности в таком случае просто увеличить размерность). В отличие от структурных значений, поля которых могут быть совершенно различными, все элементы кратного значения — одного и того же вида (см. § 6).

Рассмотрим строение паспорта кратного значения.

Ясно, что несмотря на то, что каждый элемент пронумерован одним номером, желательно сохранить понятие размерности массива и иметь возможность оперировать в программе с разным числом индексных позиций

---

\*) Заметим, что не требуется, чтобы номера шли подряд, т. е. чтобы множество номеров было отрезком ряда целых чисел. В языке могут использоваться (и используются, см. § 4 гл. III) кратные значения, в которых это условие не выполнено.

(как это было в АЛГОЛе-60). Поэтому паспорт содержит информацию о размерности, приписанной кратному значению и позволяющую поставить в соответствие набору «индексов по каждому измерению» исходный номер (или не ставить, если индексы выходят за границы измерений). Что это за информация?

Для каждого измерения нужно знать:

- а) нижнюю границу ( $l$ );
- б) верхнюю границу ( $u$ );
- в) шаг измерения ( $d$ ), т. е. изменение номера, соответствующее увеличению на единицу индекса по данному измерению. Вообще говоря, такое изменение не обязано быть постоянным, но используются только такие кратные значения, для которых это условие выполнено.

Пусть, например, имеется описание массива в АЛГОЛе-60: `array A [1:2, 1:2, 1:2]`

Тогда, если элементы массива  $A$  расположены в следующем порядке:

$A [1, 1, 1] \ A [1, 1, 2] \ A [1, 2, 1] \ A [1, 2, 2]$   
 $A [2, 1, 1] \ A [2, 1, 2] \ A [2, 2, 1] \ A [2, 2, 2]$

и номер элемента — его порядковый номер в этом ряду, то шаги измерений, считая слева направо, таковы:

$$d_1 = 4, d_2 = 2, d_3 = 1.$$

Пусть все нижние, верхние границы и шаги выписаны в виде последовательности троек:

$$(l_i, u_i, d_i), \quad i = 1, \dots, n$$

(количество троек — это и есть размерность), и пусть целые числа  $r_1, \dots, r_n$  таковы, что  $l_i \leq r_i \leq u_i$ .

Образуем число

$$I = (r_1 - l_1) \times d_1 + \dots + (r_n - l_n) \times d_n + c,$$

где  $c$  — начало отсчета — наименьший из номеров.

Если  $I$  — один из номеров (например, пусть  $r_i = l_i$ ,  $i = 1, \dots, n$ , тогда  $I = c$ ), то говорится, что паспорт описывает (*describes*) элемент с этим номером и этот элемент соответствует набору  $(r_1, \dots, r_n)$ .

Используемые кратные значения устроены так, что когда все  $r_i$  пробегает от  $l_i$  до  $u_i$ , соответствующие  $I$  пробегает все множество номеров. Фактически именно паспорт определяет допустимое множество номеров описанным здесь способом. (Это значит, что фактически

рассматриваются только обычные прямоугольные массивы, элементы которых расположены в линейном порядке.)

Число элементов таких кратных значений равно по-  
этому

$$\prod_{i=1}^n (u_i - l_i + 1), \text{ если } l_i \leq u_i, i = 1, \dots, n,$$

и нулю в противном случае.

К каждой тройке  $(l_i, u_i, d_i)$  добавляются еще две компоненты:  $i$ -е — *нижнее состояние* ( $s_i$ ) и  $i$ -е — *верхнее состояние* ( $t_i$ ), которые могут быть равными 0 или 1.

Роль  $s_i$  и  $t_i$  мы выясним в следующем параграфе.

Окончательно, таким образом, паспорт кратного значения состоит из начала отсчета  $s$  и некоторого числа  $n$  пятерок целых чисел  $(l_i, u_i, d_i, s_i, t_i)$ ,  $i = 1, \dots, n$ .

Основные преобразования кратных значений сводятся при таком их строении к тому, что производятся некоторые элементарные операции над паспортом (изменение начала отсчета, пересчет границ и шагов, исключение некоторых пятерок и т. п.), в результате чего выделяется один из элементов исходного кратного значения или создается другое кратное значение, состоящее из полученного нового паспорта и тех элементов исходного кратного значения, которые этим новым паспортом описываются.

Поначалу, правда, может показаться непонятным сам смысл вводимого аппарата преобразований, т. е. то, какие же подмассивы вырезаются из исходных массивов в каждом конкретном случае. Зато при этом достигается единообразие и строгость изложения, пояснения же можно давать особо (что мы и будем в дальнейшем делать).

#### § 4. Имена

В АЛГОЛе-60 мы привыкли расшифровывать термин «идентификатор» как «имя», «наименование». В АЛГОЛе-68 тоже есть идентификаторы, но понятие имени здесь совсем другое. Идентификаторы — это внешние объекты. *Имена* — значения, внутренние объекты, которыми, как и другими значениями, могут обладать внешние объекты, в частности, идентификаторы.

Значение, называемое именем, не считается наделенным никакими свойствами, кроме одного: оно должно

быть связано с каким-либо другим значением. Наглядно имя можно изобразить, например, кружком, от которого отходит стрелка, указывающая на произвольное значение (в частности, возможно, снова на имя).

Если стрелка имени  $N$  указывает на значение  $v$ , то говорится, что  $N$  *ссылается* на  $v$ .

Концепция имени и значения, на которое оно ссылается, является абстракцией таких общеизвестных машинных понятий, как «адрес» ячейки и ее «содержимое». Случай имени, ссылающегося на другое имя, соответствует понятию «указателя», используемого для «косвенной адресации».

Ясно, что работа с именами имеет смысл тогда, когда имеются средства, во-первых, переходить от имени к значению, на которое это имя ссылается (т. е. по адресу ячейки находить ее содержимое), и, во-вторых, менять направления стрелок (т. е. изменять содержимое ячейки с данным адресом). Первое считается заданным самим отношением ссылки, второе же есть не что иное, как аналог присваивания в АЛГОЛе-60. Действительно, связь между внешним объектом и значением (обладание значением) тоже можно было бы изобразить стрелкой, ведущей от внешнего объекта к внутреннему. В АЛГОЛе-60 присваивание значений переменным — это изменение направлений таких стрелок \*).

Если в АЛГОЛе-68 сохранить второй тип присваивания (взаимодействие между внешними и внутренними объектами), то он будет отличаться от присваивания первого типа — присваивания значения имени (изменения направления стрелки, ведущей от имени к значению, т. е. взаимодействия между внутренними объектами). Желательно же иметь лишь один механизм присваивания, поэтому в АЛГОЛе-68 отсутствует второй его тип (и, следовательно, понятие переменной, как оно есть в

---

\*) В АЛГОЛе-60 понятие переменной несколько расплывчато. С одной стороны, говорится, что идентификаторы «служат для обозначения простых переменных» и других величин, т. е. делается различие между величиной (в частности, простой переменной) и обозначающим ее идентификатором. С другой стороны, переменная определяется как «наименование, данное некоторому значению». Здесь, в случае простой переменной, под наименованием, по-видимому, понимается сам идентификатор переменной. Выбор одной из этих двух точек зрения определяет и трактовку присваивания. Поэтому трудно понять, какой из двух типов присваивания, о которых ниже говорит автор, имелся в виду в «Сообщении об АЛГОЛе-60». (Прим. ред.)

АЛГОЛе-60) и наличествует только первый. Идентификаторы начинают и перестают обладать значениями только в силу своих описаний и в течение всего периода действия этого описания обладают одним и тем же значением. Присваивание же происходит только именам.

Это совсем не умаляет общности, так как роль переменной с успехом может играть идентификатор, обладающий именем. Сам идентификатор тогда неизменно обладает этим именем, но само имя может ссылаться на разные значения. Таким образом, присваивание значения переменной в АЛГОЛе-60 моделируется присваиванием значения «верхнему» имени (т. е. имени, которым обладает идентификатор переменной). Но, сверх того, здесь возможны и присваивания именам, на которые ссылаются другие имена, что соответствует изменению цепочки косвенной адресации и не имеет прямого аналога в АЛГОЛе-60.

(Заметим, что некоторое усложнение понятия переменной по сравнению с привычным по АЛГОЛу-60 не коснется тех, кто не будет пользоваться косвенной адресацией и останется на поверхности языка, потому что конструкции языка таковы, что в данном случае можно спокойно забыть новую механику и пользоваться старой трактовкой.)

При рассмотрении строения кратных значений осталась невыясненной роль верхнего и нижнего состояний ( $s_i$  и  $t_i$ ) в пятерках паспорта ( $l_i, u_i, d_i, s_i, t_i$ ). Сейчас мы рассмотрим этот вопрос.

Предположим, что некоторое имя ссылается на кратное значение, и присваивание требует, чтобы оно начало ссылаться на другое кратное значение (той же размерности, как мы далее увидим). Границы  $l_i$  и  $u_i$  в паспорте этого другого значения, вообще говоря, могут и не совпадать с соответствующими границами первого. Однако менять границы при присваивании разрешается не всегда. Информация о том, какие границы менять можно, а какие нельзя, содержится в описании идентификатора, обладающего именем. Удобно перенести эту информацию на внутренний уровень, т. е. в значения. Эту нагрузку берет на себя кратное значение, на которое имя ссылается: для каждой из границ в тройках паспорта дополнительно задается *признак подвижности*. Этими признаками и являются состояния  $s_i$  и  $t_i$  (подвижной границе соответствует нулевое состояние, фиксированной — единичное).

При присваивании имени нового кратного значения в случае несовпадения хотя бы одной из границ проверяется, чему равно соответствующее состояние в паспорте старого кратного значения. Если оно равно единице, то дальнейшее выполнение объявляется неопределенным.

Так как состояния фактически характеризуют не само кратное значение, а ссылающееся на него имя, то и присваивание кратного значения происходит с точностью до состояний: после присваивания состояния остаются такими же, какими они были до него (т. е. содержат ту же информацию о данном имени), а состояния присваиваемого значения безразличны (они существенны лишь для своего имени, если таковое имеется).

Если на кратное значение не ссылается никакое имя, то состояния не несут никакой информации и не нужны. Но в целях единообразия состояния всегда входят в состав паспорта (в последнем случае для определенности их полагают равными единице).

Заметим, что имя, являющееся полем структурного значения (это возможно, так как любое значение может быть полем), может ссылаться на другое структурное значение (или на имя, ссылающееся на структурное значение). Это и дает возможность организовать работу со списочными структурами, о чем упоминалось в конце § 2.

## § 5. Процедурные значения (рутины)

До сих пор строение рассматриваемых внутренних объектов — значений — не имело никакого отношения к строению внешних объектов — языковых конструкций (с точностью до полей структурных значений, элементов кратных значений и значений, на которые ссылаются имена).

Значения же, называемые *рутинами* \*), наоборот, представляют собой последовательности символов, соответствующие крупному синтаксическому понятию самого АЛГОЛа-68, аналогичному составному оператору или выражению в скобках в АЛГОЛе-60. Рутинны равноправны со всеми прочими значениями в том смысле, что к ним

---

\*) Термин *рутина* (*routine*) примерно соответствует принятому в русской литературе термину *подпрограмма*. Однако с последним у программистов связано слишком много ассоциаций. Поэтому использование нового термина предпочтительнее.

применимы все общие действия над значениями: рутина может быть полем структуры, элементом кратного значения, рутину можно присвоить и т. д.

Специфическое же использование рутин сводится к следующему. Имеется класс внешних объектов (так называемые *обозначения процедур*), значения которых — рутины, формируемые путем незначительного преобразования этих внешних объектов, т. е. как бы «переводом» внешних объектов во внутренние. Далее, есть внешние объекты (аналоги операторов процедуры в АЛГОЛе-60), выполнение которых влечет «вызов» некоторой рутины (т. е. перевод внутреннего объекта во внешний), определенные ее преобразования (настройка на фактические параметры) и последующее выполнение.

Здесь существенно, что, вообще говоря, только динамически определяется, какая будет вызвана рутина, так как, например, вызов может происходить через имя, ссылающееся на рутину, а ему могут присваиваться разные рутины. В совокупности с единообразием языковых средств, позволяющих, в частности, образовывать сложные выражения, обладающие любыми значениями, в том числе и рутинami (могущими затем быть вызванными), это дает гораздо большую гибкость по сравнению с возможностями использования процедур в АЛГОЛе-60.

## § 6. Виды значений

Подразделение значений на вышерассмотренные классы понятно, так как представители различных классов — это действительно разные значения, со своей спецификой (имена могут ссылаться, рутины — вызываться, арифметические значения могут быть операндами арифметических операций и т. д.). Но этого подразделения еще недостаточно, так как и внутри одного класса имеются разные значения (арифметическое значение — это не то, что логическое, для имен существенно, на какие значения они могут ссылаться, кратные значения естественно различать по размерности, т. е. по числу пятерок в их паспорте и т. д.). Поэтому целесообразно ввести такое подразделение значений, которое учитывало бы основную специфику их использования (как мы увидим, это окажется полезным и для задания самого языка, так как можно будет явно указать, с чем и как следует работать). Другими словами, с каждым значением следует связать

некоторый качественно характеризующий его объект. В АЛГОЛе-68 эти объекты и называются *видами*.

Заметим, что различение величин в АЛГОЛе-60 по их классам и типам является отнюдь не исчерпывающим: массивы следует различать, кроме того, по их размерности, а процедуры и функции — по числу и виду параметров. Таким образом, в АЛГОЛе-60 явная концепция «полного» вида еще не обозначена. Если же попытаться ее туда ввести, то пришлось бы связывать вид с внешними объектами — идентификаторами, так как, скажем, процедуры значениями не обладают.

В АЛГОЛе-68 достаточно сопоставить виды со значениями, так как ими исчерпывается все, с чем работает язык.

Каким объектам следует поручить роль видов? Вообще говоря, это более или менее безразлично: ведь речь идет лишь о том, чтобы отобразить множество значений в некоторое другое множество и сказать, что виды значений различны, если различны их образы. Поэтому можно, например, объявить видом вещественного значения значок  $\square$ , а видом структурного значения с двумя вещественными полями — значок  $*$ . Но ясно, что это неудобно, так как такие значки никак не отражают связи между двумя значениями, а именно того, что значение вида  $*$  определенным образом сконструировано с помощью значений вида  $\square$ . Поэтому целесообразно выбирать такие множества видов, в которых подобные скелетные связи между значениями сохранялись бы и явно отражались. Каким будет состав такого множества — это уже действительно безразлично (если не считаться, конечно, с чисто психологическими соображениями), и авторы АЛГОЛа-68 выбрали тот, который был более удобен для задания синтаксиса языка \*). Как мы далее увидим, синтаксические

---

\*) Фактически множество видов языка АЛГОЛ-68 уже было определено. В самом деле, подводя итоги сказанному в §§ 1—5, приходим к следующей совокупности правил:

- 1) «логический», «литерный» и «формат» — это виды;
- 2) если  $n \geq 1$  — целое число, то «целый длины  $n$ » и «вещественный длины  $n$ » — это виды;
- 3) если  $M$  — вид, то «имя, ссылающееся на значение вида  $M$ » — это тоже вид;
- 4) если  $M, M_1, \dots, M_k$  — виды, то «процедура», «процедура, вырабатывающая значение вида  $M$ », «процедура с параметрами видов  $M_1, \dots, M_k$ » и «процедура с параметрами видов  $M_1, \dots, M_k$ , вырабатывающая значение вида  $M$ » — это тоже виды;

понятия АЛГОЛа-68 представляют собой определенные слова в алфавите малых латинских букв (не считая пробелов). Оказывается удобным, если в качестве составных частей понятий могут фигурировать виды. Поэтому виды в АЛГОЛе-68 — это тоже некоторые слова (как правило, с соответствующей мнемоникой).

Множество видов бесконечно и задается специальным образом. В следующей главе после рассмотрения соответствующих средств описательной техники мы построим это множество, а сейчас ограничимся несколькими примерами видов:

- а) *real* (вещественный);
- б) *boolean* (логический);
- в) *row of row of real* (двумерный вещественный массив);
- г) *procedure with real parameter and boolean parameter real* (вещественная функция с вещественным и логическим параметрами);
- д) *reference to reference to real* (имя, ссылающееся на имя, которое ссылается на вещественное значение).

Последние три примера показывают, как именно отражена в видах связь между соответствующими значениями. Конечно, в мнемонических целях здесь допущены некоторые излишества, можно было бы записать виды и короче, но это несущественно, тем более, что в программе соответствующие видам конструкции (описатели) более сжаты. Например, описатели, соответствующие вышеприведенным видам таковы \*):

---

5) если  $M_1, \dots, M_k$  — виды, то «структурный с полями видов  $M_1, \dots, M_k$ » — это тоже вид.

6) если  $M$  — один из видов, определенных по правилам 1—5, и  $n \geq 1$  — целое число, то «массив размерности  $n$  с элементами вида  $M$ » — это тоже вид.

Необходимы, правда, еще правила, разрешающие сомнения относительно того, может ли вид, определенный по правилу 3, 4 или 5, совпадать с одним из видов  $M, M_1, \dots, M_k$ , упоминаемых в этих правилах, или с видом, через который эти виды определены (см. § 12 гл. III, где обсуждается эта проблема).

Для множества видов, которое можно определить таким способом, надо выбрать представление, позволяющее описать любой конкретный вид. Фактически нужны два представления — одно в метаязыке (см. § 2 гл. II), с помощью которого описывается синтаксис языка АЛГОЛ-68, другое — в самом языке для описания видов значений, которыми могут обладать идентификатор и некоторые другие внешние объекты. (Прим. ред.)

\*) Здесь используется второе из тех представлений множества видов, о которых шла речь в предыдущем примечании, (Прим. ред.)

- а) real    б) bool
- в) [,] real    г) proc (real, bool) real
- д) ref ref real

Итак, каждому значению приписывается некоторый вид. Наглядно, таким образом, значение можно рассматривать как «учетную карточку», на которой написаны, во-первых, вид, и, во-вторых, само значение этого вида.

## § 7. Области действия значений

В АЛГОЛе-60 термин «область действия» связывается с внешними объектами — идентификаторами. Например, блок, в заголовке которого идентификатор описан — его область действия. Говорится, что объект, обозначаемый идентификатором, действует только в области действия этого идентификатора, а вне ее пропадает. Каждый идентификатор должен иметь область действия (в частности, идентификаторы — не формальные параметры и не метки — должны быть описаны).

Такое понятие области действия несет двойную нагрузку. Во-первых, оно связано с обработкой языковых конструкций: требуется, чтобы для каждого употребляемого идентификатора имелось его вхождение, ответственное за появление объекта, обозначаемого этим идентификатором. (В АЛГОЛе-68 такие вхождения называются *определяющими*, в отличие от других вхождений, называемых *примененными*.) Эта часть понятия области действия работает, таким образом, на уровне внешних объектов.

Вторая часть этого понятия связана с собственно появлением и уничтожением объектов, обозначаемых идентификаторами (и при реализации сводится в конечном счете к распределению и использованию памяти).

(Различие этих двух сторон проявляется при работе с собственными величинами: идентификатор, снабженный описателем *own*, имеет обычную область действия, но при выходе из нее величина, им обозначаемая, фактически не пропадает, так как при повторном входе требуется ее восстановление.)

В АЛГОЛе-68 объекты, обозначаемые идентификаторами, — это значения, которыми идентификаторы обладают (т. е. внутренние объекты), и поэтому «внешняя» и «внутренняя» стороны старого понятия области действия здесь разделены.

Внешняя сторона выглядит следующим образом. Даются правила, по которым примененные вхождения отличаются от определяющих и для каждого примененного вхождения идентификатора отыскивается соответствующее определяющее. (Эти правила аналогичны тем, по которым в АЛГОЛе-60 разыскивается описание идентификатора по уровням локализации от самого внутреннего, содержащего рабочее вхождение идентификатора, к внешним.) Говорится, что примененное вхождение *идентифицирует* найденное определяющее, и требуется, чтобы всякое примененное вхождение идентификатора идентифицировало определяющее. Это полностью аналогично АЛГОЛу-60 (хотя там правила идентификации явно не сформулированы), но термин «область действия» к идентификаторам не применяется. Этот термин связывается только со значениями. А именно, каждому используемому значению (т. е. значению, которым обладают какие-либо внешние объекты — части данной программы) ставится в соответствие некоторая часть программы, и эта часть называется *областью действия* значения.

Язык устроен так, что любым данным значением могут обладать только те идентификаторы (или другие внешние объекты), которые входят в область действия этого значения. Если выполнение области действия какого-либо значения завершено, то все внешние объекты, ранее обладавшие этим значением, перестают им обладать, так что можно считать, что значение пропадает.

Правда, здесь есть одна тонкость, связанная с именами. Предположим, что имя  $N_1$  ссылается на другое имя  $N_2$ , которое в свою очередь ссылается на некоторое значение  $v$ . Пусть далее именем  $N_1$  обладает идентификатор И. Тогда в программе, манипулируя с И, можно, в частности, воздействовать и на  $v$ , так как имеется цепочка  $N_1 \rightarrow N_2 \rightarrow v$ . Но как быть, если область действия  $N_1$  больше области действия  $N_2$  (т. е. объемлет ее) и эти манипуляции происходят внутри области действия  $N_1$  (а это законно, так как И еще может обладать  $N_1$ ), но вне области действия  $N_2$ ? Ведь если мы считаем, что  $N_2$  пропало, то цепочка, ведущая от  $N_1$  к  $v$ , разорвана. Если же отказаться от пропадания значений, то это приведет к загромождению памяти при реализации и сделает бессодержательным само понятие области действия (можно будет считать, что область действия любого значения — вся программа). Авторы языка выходят из положения «декретивным»

путем. Как уже говорилось, отношение ссылки устанавливается посредством присваивания значения имени. Присваивание же разрешается выполнять в том случае, если область действия имени не больше области действия присваиваемого значения, в противном случае выполнение присваивания объявляется неопределенным. Поэтому в нашем примере имени  $N_1$  нельзя было присвоить  $N_2$ , и поэтому  $N_1$  не может ссылаться на  $N_2$ .

Рассмотрим теперь, какие области действия сопоставляются с различными значениями.

Областью действия простых значений считается вся программа, т. е. считается, что все простые значения появляются перед началом выполнения программы и не уничтожаются до завершения ее выполнения. (Аналогией здесь может служить поле констант в машинных программах.)

Содержательно область действия определяется для имен. Область действия имени — это некоторая часть программы, называемая *уровнем*, аналогичная внутренности блока или телу процедуры в АЛГОЛе-60 и содержащая определяющее вхождение идентификатора, обладающего данным именем. Имена создаются при выполнении их уровня и исчезают при завершении его выполнения. (Здесь аналогией могут служить рабочие ячейки, которые заводятся на время выполнения какой-либо части машинной программы.) Какие именно конструкции являются для имен уровнями, будет рассмотрено в главе III.

Областью действия рутины считается наименьший уровень из тех, которые содержат определяющие вхождения всех тех идентификаторов, чьи примененные вхождения содержатся в рутине (т. е. наименьший уровень, внутри которого рутина может вызываться и выполняться)

Наконец, областью действия кратного или структурного значения считается наименьшая из областей действия значений, входящих в его состав.

Таким образом, содержательность понятия области действия сводится к областям действия имен, все же остальное, могущее быть связанным с областями действий значений, — простое логическое следствие. Это соответствует тому, что именно появление и уничтожение имен связывается при реализации с использованием рабочей памяти.

## § 1. Строгий язык и язык представлений.

## Синтаксис строгого языка

В АЛГОЛе-68, так же как и в АЛГОЛе-60, законченные, подлежащие выполнению конструкции, называются *программами*, а вообще конструкции представляют собой последовательности символов. Однако с символами связан некоторый трюк. Мы привыкли, что символ — это неделимый значок. Строение значков несущественно, они только должны отличаться друг от друга. Но для способа, которым задан синтаксис АЛГОЛа-68 и о котором речь впереди, произвольные значки-символы оказываются неудобными и авторы вводят другое понятие символа, называя *символом* произвольную последовательность малых латинских букв, оканчивающуюся словом *symbol*. Друг от друга такие символы отделяются запятой. Таким образом, потенциально символов оказывается бесконечно много, но по ходу описания языка вводится лишь конечное число фактически используемых символов, например, *comma symbol*, *letter a symbol* и другие (пробелы в символах роли не играют).

Ясно, что писать программу в таких «символах» — дело невеселое, поэтому авторы поступают следующим образом. Каждому используемому символу сопоставляется некоторый значок, называемый *представлением* этого символа. Конструкция, записанная в исходных символах (последовательностях малых латинских букв, разделенных запятыми), — это конструкция *строгого языка*. Если заменить в ней символы на их представления и убрать все разделявшие их запятые, то получается конструкция *языка представлений*. Синтаксис АЛГОЛа-68 — это синтаксис строгого языка, программы же пишутся, конечно, на языке представлений, т. е. в обычных символах.

Например, для символов *letter a symbol*, *letter b symbol* и *becomes symbol* даются соответственно представления  $a$ ,  $b$  и  $:$ . Поэтому если в конструкции языка представлений имеется вхождение

$$a : = b$$

то в соответствующей конструкции строгого языка в этом месте будет

*letter a symbol, becomes symbol, letter b symbol*

Синтаксис строгого языка представляет собой совокупность *порождающих правил*, по смыслу и строению аналогичных металингвистическим формулам АЛГОЛа-60. Принципиальное отличие от АЛГОЛа-60 заключается в том, что множество порождающих правил бесконечно и потому само порождается с помощью *метаязыка* (синтаксис которого уже конечен). Как и для чего это сделано, мы рассмотрим в следующем параграфе, сейчас же остановимся на строении самих порождающих правил.

Каждое такое правило состоит из двух частей, разделенных двоеточием. Левая часть представляет собой последовательность малых латинских букв (слово), правая либо пуста, либо тоже слово, либо нескольких слов, разделенных запятыми. Между буквами могут стоять пробелы, которые считаются несущественными. После такого правила ставится точка (свидетельствующая о конце правой части).

Слово называется *понятием*, если оно стоит в левой части какого-либо правила (обычно понятия — это английские выражения с соответствующей мнемоникой). Тогда правая часть этого правила называется *прямым порождением* этого понятия. Прямое порождение некоторого понятия считается также его *порождением*. Если какое-либо слово из некоторого порождения понятия  $A$  есть понятие  $B$ , то, заменив это слово на любое прямое порождение понятия  $B$ , получим новое порождение понятия  $A$ . Такую операцию можно проделывать любое число раз.

*Окончательным* (или *терминальным*) порождением понятия называется либо пустое порождение, либо порождение, состоящее только из символов (т. е. слов, оканчивающихся на *symbol*, разделенных запятыми, если их больше одного). Например, все программы АЛГОЛа-68 в строгом языке — окончательные порождения понятия *program*.

Несколько порождающих правил с одним понятием в левой части могут быть объединены в одно. Тогда правая часть такого правила состоит из правых частей исходных правил, разделенных точкой с запятой. После объединенного правила также ставится точка.

Мы видим, что схема здесь в общем такая же, как и в АЛГОЛе-60 (хотя там и нет таких четких определений). Понятия соответствуют металингвистическим переменным, символы — основным символам (знакам), двоеточие — знаку  $::=$ , точка с запятой — связке  $|$ .

Запятая здесь служит для отделения друг от друга символов и понятий. В АЛГОЛе-60 она не нужна, так как символы — значки, а металингвистические переменные (понятия) заключаются в угловые скобки. Здесь, наоборот, не нужны эти скобки.

Удобство именно такого строения правил связано со способом их порождения с помощью метаязыка, к рассмотрению которого мы сейчас приступаем.

## § 2. Метаязык и использование метапонятий для образования синтаксиса строгого языка. Виды

Как уже говорилось, множество порождающих правил синтаксиса строгого языка бесконечно и образуется с помощью метаязыка. Такое усложнение описательной техники по сравнению с одноступенчатым заданием синтаксиса в АЛГОЛе-60 связано со стремлением исключить из множества всех конструкций, которые формально законны в языке (т. е. таких, которые порождаются синтаксисом, и, возможно, удовлетворяют дополнительным формализованным условиям) как можно больше тех конструкций, которые не имеют однозначной семантической интерпретации. (В идеале, конечно, хотелось бы, чтобы синтаксисом порождалось бы только то, что нужно, но это очень трудная задача.)

В АЛГОЛе-60 в этом смысле имеется большое несоответствие, связанное в основном с видами величин. Происходит это, в частности, потому, что эти виды никак не отражены в формулах синтаксиса.

Например:

$\langle \text{оператор процедуры} \rangle ::= \langle \text{идентификатор процедуры} \rangle$   
 $\quad \quad \quad \langle \text{совокупность фактических параметров} \rangle$   
 $\langle \text{фактический параметр} \rangle ::= \langle \text{выражение} \rangle | \dots$

Кто посмеет на уровне только такого формализма бросить камень, скажем, в следующее описание процедуры:

**procedure  $p$  ( $f$ ); procedure  $f$ ; begin  $f$  (1);  $f$  (true) end**

Ведь никаких других условий, которые надо дополнительно проверять, синтаксис языка не содержит. (Некоторые такие условия весьма приблизительно описаны в семантических разделах.)

Ясно, что приведенную простую ситуацию можно сколь угодно глубоко запутать, например, взяв в качестве фактических параметров сложные выражения, сами содержащие несогласования видов и т. д. Правда, дополняя АЛГОЛ-60 явной концепцией вида, удастся формализовать условия согласования и дать алгоритмы выбраковки не удовлетворяющих им конструкций (см. [4]), однако это оказывается гораздо сложнее, чем можно было бы ожидать. Дело в том, что в АЛГОЛе-60 нет средств описывать все «полные» виды. (В нашем примере в заголовке процедуры отсутствует информация, сколько параметров у  $f$  и какого они вида.) Поэтому необходимую информацию приходится, вообще говоря, накапливать из рабочих (т. е. примененных) вхождений идентификаторов, а это приводит к значительному усложнению соответствующих формализмов и методов анализа.

В АЛГОЛе-68 в этом направлении сделано следующее.

Во-первых, определяющее вхождение идентификатора всегда содержит всю информацию о его виде (т. е. о виде обладаемого им значения), что в совокупности с правилами идентификации определяющего вхождения снимает вышеуказанную трудность (хотя за это приходится платить громоздкостью описателей в случае сложных видов).

Во-вторых, всюду, где в порождающих правилах требуется по смыслу согласование видов, в состав этих правил определенным образом входят сами согласуемые виды. Таким образом, вместо одного правила, которое было бы на уровне описательной техники АЛГОЛа-60, появляется столько правил, сколько можно получить, варьируя все допустимые в данном случае виды.

Этим сразу же все становится на место. Действительно, в терминах нашего примера появятся правила, где будет фигурировать не просто «идентификатор процедуры», но «идентификатор процедуры с такими-то параметрами», и эти «такие-то параметры» будут в правилах определенным образом встречаться (как именно — увидим далее).

Поэтому, так как определяющее вхождение доставит всю информацию о параметре  $f$ , то хотя бы один из двух операторов окажется неверным (не будет порождаться правилом, соответствующим виду  $f$ ). (На самом деле все обстоит несколько сложнее, но основная идея базируется на такой схеме.)

Поскольку видов бесконечно много, то и правил, соответствующих одному безвидовому правилу, часто оказывается бесконечно много. Поэтому если принять такой способ согласования видов, то возникает вопрос, с помощью чего можно все эти правила задать и как это сделать. Способ, принятый в АЛГОЛе-68, состоит в следующем. Представим себе, что каким-то образом заданы и обозначены все множества, элементы которых фигурируют в правилах синтаксиса. Тогда можно поступить так:

1) вместо всех правил, получающихся варьированием элементов каких-либо из этих множеств, написать одно «параметрическое» правило, которое получится, если на места, занимаемые элементами множеств, подставить обозначения этих множеств;

2) объявить, что всякое правило получается из параметрического заменой обозначений множеств какими-либо их элементами (причем одно и то же обозначение заменяется одним элементом, а представители разных никак не связаны). Так как обозначенные множества считаются заданными, то тем самым заданы и все правила.

Но как задать такие бесконечные множества? Для этого можно использовать уже привычное средство — порождающие правила некоторого другого языка, который и будет называться метаязыком. Тогда понятия этого языка (*метапонятия*), порождающие нужные множества, могут служить и обозначениями этих множеств в параметрических правилах.

Такая схема и принята в АЛГОЛе-68. Рассмотрим подробнее, как она реализована.

Роль метапонятий исполняют слова, состоящие из больших латинских букв. Это тоже обычно английские слова с соответствующей мнемоникой. Например, все виды порождаются метапонятием *MODE* (*mode* — вид). В отличие от понятий пробелы между буквами здесь не допускаются и служат для отделения метапонятий друг от друга. (Это удобно, так как в параметрических правилах можно ставить рядом несколько метапонятий, разделяя их лишь пробелами. При замене их частями понятий про-

белы перестают играть роль разделителей, и все «сливается в единое слово».)

Порождающие правила метаязыка (*метаправила*) устроены аналогично правилам строгого языка. Метаправило состоит из двух частей, разделенных двоеточием. В левой части находится метапонятие, правая либо пуста, либо тоже метапонятие, либо слово, состоящее из малых латинских букв (но не обязательно понятие строгого языка), либо, наконец, несколько метапонятий или таких слов, разделенных пробелами. После метаправила тоже ставится точка.

Правая часть метаправила называется *прямым порождением* метапонятия, стоящего в левой части. Прямое порождение некоторого метапонятия считается также его *порождением*. Если какое-либо слово из некоторого порождения метапонятия *A* есть метапонятие *B*, то, заменив это слово на любое прямое порождение метапонятия *B*, получим новое порождение метапонятия *A*. Такую операцию можно прodelывать любое число раз.

*Окончательным порождением* метапонятия называется такое его порождение, которое не содержит метапонятий, т. е. слово (возможно, пустое), состоящее только из малых латинских букв и пробелов.

Окончательные порождения метапонятий и являются теми частями понятий, которыми заменяются эти метапонятия в параметрических правилах (после чего пробелы в них перестают играть роль).

Несколько метаправил с одним метапонятием в левой части тоже могут быть объединены тем же способом, что и правила строгого языка.

Таким образом, механизм порождения в метаязыке совершенно аналогичен тому, что мы видели в самом языке. Но поскольку метаязык носит вспомогательный характер, и его конструкциям (окончательным порождениям метапонятий) не требуется давать семантической интерпретации, то никаких вопросов типа «согласования чего-то с чем-то» здесь не возникает, и порождается все то, что нужно для технических целей. Поэтому синтаксис метаязыка конечен (и относительно невелик) и только нотацией отличается от бэкусовско-науровской формы, которой задан АЛГОЛ-60.

Рассмотрим в качестве примера упрощенный синтаксис аналога оператора процедуры в АЛГОЛе-60, понятия *call* — вызов (рутины):

*call: procedure with PARAMETERS primary,  
open symbol, actual PARAMETERS, close symbol.*

Разберемся, что здесь написано.

Во-первых, *open symbol* и *close symbol* — это в языке представлений просто скобки ( и ), в которые, таким образом, заключаются фактические параметры.

Метапонятие *PARAMETERS*, как мы далее увидим, таково, что каким бы окончательным порождением его ни заменить, все, что написано между : и *primary*, будет видом (т. е. частным случаем окончательного порождения *MODE*), начинающимся с *procedure with* (далее мы приведем метаправила, связанные с *MODE*).

В языке имеется параметрическое правило, в левой части которого стоит *MODE primary* (первичное выражение такого-то вида). Это правило порождает соответствующие конструкции (последовательности символов). Тем самым все, что стоит до открывающей скобки (*open symbol*), синтаксически определено. (Но заметим, что такую позицию может занимать не только идентификатор, как это было в АЛГОЛе-60, но любое первичное выражение нужного вида. Первичным же, как и в АЛГОЛе-60, можно сделать любое выражение, заключив его в скобки.)

Прежде чем идти дальше, рассмотрим правила метаязыка, связанные с метапонятием *PARAMETERS*.

*PARAMETERS: PARAMETER; PARAMETERS and  
PARAMETER.*

*PARAMETER: MODE parameter.*

Пока что это ничего не прояснило, так как еще непонятно, что делать с *MODE parameter*. Но отсюда уже видно, как согласуются виды первичного выражения, стоящего перед скобками (в частности, идентификатора), и конструкции, выступающей в роли совокупности параметров — ведь при образовании конкретных правил два вхождения *PARAMETERS* заменятся одним и тем же его окончательным порождением.

Пойдем дальше. В языке имеется правило, в левой части которого стоит *actual MODE parameter* (порождающее конструкции, могущие быть фактическими параметрами нужного вида). Дадим теперь следующее правило:

*actual PARAMETERS and PARAMETER:  
actual PARAMETERS, comma symbol,  
actual PARAMETER.*

Предоставляем убедиться, что теперь синтаксически определено и то, что заключено в начальном правиле в скобки (*comma symbol* — это запятая в языке представлений).

Мы привели сравнительно простой пример взаимодействия языка и метаязыка, причем и его сознательно упростили. Но даже этот пример с непривычки может показаться устрашающим. Действительно, свыкнуться с такой техникой не очень просто, тем более, что совокупность правил образует большое и связное целое, в котором все тесно переплетено. Но ведь и техника АЛГОЛа-60 поначалу вызывала большие затруднения, так что все дело здесь в том, чтобы «набить руку». Исходя из собственного опыта, мы не считаем эту задачу особенно трудной.

Рассмотрим теперь, какие различаются виды. Как уже говорилось, вид — это окончательное порождение метапонятия *MODE*. Мы определим *MODE* несколько упрощенно по сравнению с [2] с целью не утомлять читателя деталями, не являющимися принципиальными с «видовой» точки зрения, а также не вводить сейчас концепции так называемого *объединенного* вида, которая будет отдельно рассмотрена в главе V.

*MODE: TYPE; STOWED.*

*TYPE: PLAIN; format; PROCEDURE;*  
*reference to MODE.*

*PLAIN: INTREAL; boolean; character.*

*INTREAL: INTEGRAL; REAL.*

*INTEGRAL: LONGSETY integral.*

*REAL: LONGSETY real.*

*LONGSETY: EMPTY; long LONGSETY.*

*EMPTY:*

*PROCEDURE: procedure PARAMETY MOID.*

*PARAMETY: with PARAMETERS; EMPTY.*

*PARAMETERS: PARAMETER;*

*PARAMETERS and PARAMETER.*

*PARAMETER: MODE parameter.*

*MOID: MODE; void.*

*STOWED: structured with FIELDS; row of MODE.*

*FIELDS: FIELD; FIELDS and FIELD.*

*FIELD: MODE field.*

Предоставляем убедиться, что примеры видов, ранее приведенные в § 6 гл. I, действительно являются видами, т. е. порождаются метапонятием *MODE*.

Виды простых значений — окончательные порождения *PLAIN* (*plain* — простой), процедурные виды порождаются *PROCEDURE*. Виды кратных значений начинаются с *row of*, структурных — со *structured with*, имен — с *reference to*.

Усложнение видов вещественных и целых значений связано с тем, что различие их длин считается видовым, и поэтому требуется иметь разные виды для таких значений. Это достигается возможностью многократного приписывания *long* (длинный) к видам *real* и *integral*.

Если вид процедурного значения заканчивается словом *void* (пустой), это означает, что речь идет о процедуре, не являющейся функцией, в противном случае — о функции того вида, который стоит вместо *void*. (Вообще, когда *void* стоит на месте вида, это всегда имеет аналогичный смысл. Наличие специального слова вместо просто отсутствия вида оказывается синтаксически удобным, но, конечно, непринципиально.)

В остальном предлагаем разобраться в приведенных правилах самостоятельно.

В заключение настоящего параграфа заметим, что не все метапонятия, встречающиеся в параметрических правилах языка, имеют бесконечно много окончательных порождений, есть и такие, у которых их конечное число. Принципиальной необходимости иметь такие метапонятия нет: можно явно выписать все правила, содержащие их окончательные порождения. Роль этих метапонятий, таким образом, сводится просто к сокращению записи. Поэтому при изучении метаязыка полезно различать, что в него попало по существу дела, а что привнесено с целью сокращений. (Интересно, что некоторые вещи, выступавшие в [1] на уровне языка, в [2] вынесены в метаязык, так как это позволило сократить описание.)

### § 3. Действия

В АЛГОЛе-68 процесс выполнения различных языковых конструкций (в частности, программ) формулируется (в разделе «Семантика») в терминах некоторого воображаемого *устройства* (или *вычислителя*, или *машины*, можно назвать как угодно, так как этот термин — первичный, и за ним не скрывается ничего, кроме того, что в него вкладывается в дальнейшем). Считается, что это устройство способно производить определенные *действия* над *объектами*.

*Объекты* — это внешние объекты (части программ) и внутренние (значения).

*Действия* в конечном счете сводятся к установлению или устранению *отношений* между объектами. Например,

«обладать» и «ссылаться» — отношения. Мы не будем рассматривать сейчас всех вводимых в языке отношений и действий, они будут появляться по ходу изложения семантики \*). Отметим только, что имеются действия, называемые *последовательными* и *параллельными*.

Последовательные и параллельные действия состоят из некоторых других действий (которые тоже могут быть последовательными или параллельными).

Выполнение *последовательного* действия состоит из последовательного (т. е. в определенном порядке) выполнения его компонент. Такое выполнение привычно по АЛГОЛу-60. Например, выполнение составного оператора в АЛГОЛе-60 можно рассматривать как последовательное действие, состоящее из выполнения его подоператоров (могущих тоже быть составными).

Интересна концепция параллельного действия, отсутствующая в явной форме в АЛГОЛе-60. Выполнение *параллельного* действия состоит из выполнения его компонент в произвольном, не определенном в языке порядке (так что можно считать, что все эти компоненты выполняются «одновременно»). Наличие таких действий дает следующее.

Во-первых, имеется возможность описывать параллельные процессы (например, с помощью так называемых *параллельных выражений*). Разумеется, не всякие действия можно выполнять параллельно, так как порядок их выполнения может оказаться существенным. Но если параллельное выполнение некоторых действий возможно (а такое бывает весьма часто, например, в тексте « $x := 0$ ;  $y := 1$ » безразлично, в каком порядке выполнять операторы), то явное указание на это может повысить эффективность программы при использовании машин с возможностями совмещенного выполнения.

Во-вторых, сразу же перекладывается на плечи авторов программ ответственность за использование всякого рода побочных эффектов. А именно, при выполнении определенных конструкций действие, состоящее из выполнения некоторых частей этих конструкций, объявляется параллельным. Например, параллельным объявляется

---

\*) Строго говоря, можно поставить вопрос: а осуществимо ли вообще устройство, способное выполнить все то, что от него требует АЛГОЛ-68? Авторы не занимались специальным доказательством этого, но мы далее увидим, что все способности устройства алгоритмичны и могут быть промоделированы,

выполнение левой и правой части присваивания (а это могут быть весьма сложные выражения), так что если одна из этих частей содержит побочный эффект, действующий на другую, то результат окажется неопределенным (так как он зависит от порядка выполнения), но автор программы об этом предупрежден.

Заметим, что так как компонентой параллельного действия может быть снова параллельное действие или последовательное действие, содержащее параллельное, то процесс распараллеливания может идти сколь угодно глубоко, и взаимный порядок выполнения компонент на всех уровнях определен только с точностью до порядка, диктуемого последовательными действиями.

#### § 4. Расширения и расширенный язык

В конце описания АЛГОЛа-68 (уже после формулирования его синтаксиса и семантики) приводится серия пунктов, в каждом из которых разрешается некоторые (как правило, часто встречающиеся) комбинации конструкций записывать более коротко, чаще всего опуская какие-либо их части, содержащие избыточную информацию. Эти пункты называются *расширениями*, а конструкции, записанные с использованием расширений — это конструкции *расширенного языка*.

Может возникнуть вопрос: а почему же с самого начала синтаксис не задан таким образом, чтобы конструкции сразу приняли окончательный вид? Это было бы неудобным, так как АЛГОЛ-68 построен так, что его конструкции, как правило, скомбинированы из составных частей, имеющих свой синтаксис и семантику, и семантика всей конструкции формулируется в терминах этих «кирпичей». «Расширенные» же конструкции представляют собой сокращенную запись какого-либо частного случая нерасширенных, причем информация о блочности строения в такой сокращенной записи утеряна. Следовательно, если расширенные конструкции будут сразу появляться в языке в окончательном виде, то для них придется дополнительно формулировать семантику, причем для каждой в своих терминах, так как такие конструкции будут представлять собой неделимое целое. Здесь же этого делать не нужно, так как явно указано, каким частным случаем чего более общего (имеющего свою общую семантику) является каждая конструкция расширенного языка.

Кроме вышесказанного, расширения играют еще и другую роль. В описании самого языка вошло только его «ядро», система независимых понятий, не сводящихся (по крайней мере достаточно просто) друг к другу. Все же остальное, т. е. то, что целесообразно включить в язык, можно выразить уже имеющимися в нем средствами, вынесено из ядра и включено в стандартные описания и, частично, в расширения. (Набор стандартных описаний также приводится.)

Например, в АЛГОЛе-68 имеется оператор, обобщающий оператор цикла в АЛГОЛе-60. Но появляется он только в расширениях: в одном из пунктов приводится конструкция, представляющая собой трактовку этого оператора средствами ядра, и говорится, что эта конструкция может быть записана так (в самом общем случае):

**for  $I$  from  $A$  by  $B$  to  $C$  while  $D$  do  $E$**

где буквы обозначают соответствующие конструкции. Далее перечисляется, в каких случаях что можно опускать. Одной из самых распространенных форм записи будет такая:

**to  $n$  do  $E$**

(от 1 до  $n$  с шагом 1, параметр цикла в  $E$  не используется).

Самая короткая из возможных записей такая:

**do  $E$**

(повторять, пока не будет перехода из  $E$  вне  $E$ ).

Таким же путем вводятся еще некоторые удобные средства. Мы не будем останавливаться на том, что именно и когда разрешается, это не столь уж существенно, но некоторые примеры расширений приводить будем, так как иначе у читателя может создаться неоправданное впечатление громоздкости языка.

На этом мы закончим рассмотрение элементов описательной техники АЛГОЛа-68 и перейдем к главному: разбору его конструкций.

## ОСНОВНЫЕ КОНСТРУКЦИИ

Мы начнем рассмотрение конструкций АЛГОЛа-68 с самых первичных (аналогов первичных выражений и основных операторов в АЛГОЛе-60). Такие конструкции называются *базами* (*bases*). Разумеется, в силу рекурсивного строения языка в состав баз могут входить и другие, сколь угодно сложные конструкции, но эта ситуация уже привычна по АЛГОЛу-60 (например, индекс в переменной с индексами может быть произвольным арифметическим выражением).

Мы слегка упростим рассмотрения, связанные с базами, в частности их синтаксис. Дело в том, что в языке имеется конструкция, которая считается базой, только будучи заключена в скобки. Такое усложнение синтаксиса и нарушение его единообразия вызвано отнюдь не принципиальными соображениями и даже не страховкой от синтаксических неоднозначностей, а только тем, что при выбранных представлениях некоторых символов отсутствие скобок может привести на практике к двусмысленной трактовке их комбинаций на языке представлений (но не комбинаций самих символов на строгом языке). Для наших целей это, конечно, несущественно, и мы произведем соответствующие упрощения.

Различаются базы определенного вида, т. е. обладающие значениями этого вида (например, 1 — вида *integral*) и не обладающие никакими значениями (как операторы в АЛГОЛе-60). Ниже приведены соответствующие правила (все правила мы будем давать без перевода, рассчитывая на элементарное знакомство читателя с английским языком):

*MODE base: MODE mode identifier; MODE denotation;  
MODE call; MODE slice.*

*void base: void call.*

Таким образом, в строгом языке есть не просто понятие *base*, но, например, *real base*, *reference to boolean base* и т. п.

Следующие четыре параграфа посвящены разбору баз.

## § 1. Идентификаторы

Как видно из правой части первого из вышеприведенных правил, в языке есть не просто понятие *identifier* (идентификатор), но, например,

*real mode identifier, reference to boolean mode identifier* и т. п. (Подобная ситуация будет часто встречаться в дальнейшем, и мы больше не будем ее особо оговаривать.) Но все эти понятия порождают одно и то же — обыкновенные идентификаторы, привычные по АЛГОЛу-60 (т. е. конечные последовательности букв и цифр, начинающиеся с буквы). Для чего же нужно иметь несколько (в данном случае даже бесконечно много) понятий, порождающих одно и то же?

Дело в том, что по ходу выполнения программы становится известным вид, связанный с идентификатором, т. е. каким именно понятием он порожден. Это позволяет разворачивать программу дальше, перенося информацию на объемлющую конструкцию, для которой, таким образом, тоже становится известным, каким понятием она порождена (ситуация, когда понятия, связанные с разными видами, имеют много одинаковых порождений, типична), и т. д. до завершения выполнения какой-либо крупной синтаксической единицы.

Выполнение идентификатора (это звучит несколько странно, но термин «выполнение» равно принят по отношению ко всем конструкциям) заключается в отыскании (по правилам идентификации) его определяющего вхождения (здесь и становится известным его вид) и установлении отношения «обладать» между данным вхождением идентификатора (в позиции базы оно всегда примененное) и значением, которым обладает его определяющее вхождение.

## § 2. Обозначения

Конструкции, называемые *обозначениями* (*denotations*) — это «константы» определенного вида. Например, 1 — обозначение целого, обладающее значением вида *integral* (а именно, единицей). Каждое обозначение обладает фиксированным значением, не зависящим от выполнения программы.

Интересно, что, кроме привычных обозначений таких, как 1, *true* и т. п., имеются обозначения процедур, значения которых — рутин. С одной стороны, такие

обозначения, как и любые другие, можно рассматривать как константы и обращаться с ними так же, как и с обозначением единицы (единственное отличие в том, что такая «единица» — другого вида, так что ее значение арифметически сложить с чем-то нельзя, но, например, присвоить можно). С другой же стороны, обозначения процедур представляют собой языковые конструкции, из которых определенным образом формируются рутинны — их значения.

Выполнение обозначения состоит в том, что ничего не делается. Осуществимость «ничегонеделания», по-видимому, не вызывает сомнений.

Заметим, что хотя в правиле для баз фигурирует *MODE denotation*, обозначения определяются не для всех видов. Вышеупомянутое вхождение *MODE* просто удобно, так как *MODE* еще несколько раз входит в правило (и уже на полных правах, как, например, в случае идентификаторов). Поэтому, например, *reference to real denotation* не есть понятие строгого языка и не имеет порождений (обозначения имен отсутствуют, ибо в них нет нужды по самому смыслу работы с именами, как просто с указателями на другие значения). Такая ситуация называется *тупиком* и встречается довольно часто.

Рассмотрим, какие различаются обозначения:

- \* *denotation*: *PLAIN denotation*;  
                  *row of character denotation*;  
                  *BITS denotation*;  
                  *procedure with PARAMETERS MOD*  
                  *denotation*.

Приведенное правило является в языке вспомогательным, оно служит только для удобства перечисления. На самом деле понятие *denotation* (без вида) нигде не используется. В таких случаях слева от правила ставится звездочка.

Разберемся сначала в *простых обозначениях*, порождаемых *PLAIN denotation*, для чего достаточно задать обозначения для всех видов, являющихся окончательными порождениями *PLAIN* (напомним, что это — виды *real* и *integral*, к которым слева можно приписывать *long* сколько угодно раз, а также виды *boolean* и *character*).

Правила для *real denotation* и *integral denotation* приводят к почти такой же записи вещественных и целых чисел, как в АЛГОЛе-60. Мы опустим эти правила, а также их семантику, в которой указывается, какими значениями обладают эти обозначения, так как речь здесь идет о привычных вещах.

Правило, связанное с «длинными» числами, таково:  
*long INTREAL denotation: long symbol, INTREAL denotation.*

Таким образом, числа длины  $n$  обозначаются путем приписывания  $n - 1$  раз символа **long** (представления *long symbol*) к обозначениям обычных чисел (т. е. чисел единичной длины). Например:

**long 0**

**long long 3.14159265358979323846**

Значением обозначения длины  $n$  является пара (число,  $n$ ), где число — такое же, как у соответствующего обозначения длины 1, т. е. получающегося из данного обозначения отбрасыванием всех символов **long** (в реализации значениями обозначений разных длин будут числа, представленные различными способами).

Требуется, чтобы число не превосходило диапазона длины  $n$  (см. § 1 гл. 1), в противном случае значение обозначения объявляется неопределенным.

Таким образом, если реализация АЛГОЛа-68 такова, что используемые числа не выходят за ее диапазон единичной длины, и точности, поставляемой этой длиной, достаточны для некоторой программы, то при написании этой программы можно спокойно забыть про существование **long**. В противном же случае придется, манипулируя с **long**, повышать требования к точности вычислений (что, однако, возможно только в пределах, предусмотренных реализацией, так как, начиная с некоторой максимальной длины, диапазоны считаются постоянными).

Рассмотрим оставшиеся два простых обозначения:

*boolean denotation: true symbol; false symbol.*

Семантика очевидна — значение *true symbol* (**true**) есть *true*, значение *false symbol* (**false**) — *false*.

*character denotation: quote symbol, string item, quote symbol.*

*string item: character token; quote image.*

*quote image: quote symbol, quote symbol.*

Здесь *quote symbol* — это кавычка, для которой дается представление", понятие *character token* (литерный знак) порождает определенные символы (в частности, буквы и цифры). Считается, что каждый такой знак обладает фиксированным литерным значением (можно считать, что эти значения — копии самих литерных знаков с точностью до типографских особенностей). Таким образом, литерное

обозначение представляет собой либо литерный знак, заключенный в кавычки (например, "a"), либо «двойную кавычку», заключенную в кавычки (т. е. четыре кавычки подряд: """). Считается, что двойная кавычка тоже обладает фиксированным литерным значением (которое можно рассматривать как копию самой кавычки).

Литерное обозначение обладает таким же значением, что и его литерный знак или двойная кавычка.

Роль литерных обозначений совершенно аналогична роли числовых или логических. Если  $x$  — вида *reference to integral*, а  $y$  — вида *reference to boolean*, то ясен смысл, например, следующих присваиваний (хотя, строго говоря, о присваиваниях нам еще ничего неизвестно):

$$\begin{aligned} x &:= 1 \\ y &:= \text{true} \end{aligned}$$

Точно так же, если  $z$  — вида *reference to character*, то можно написать  $z := "a"$  или  $z := ""$ . (Отсюда ясна необходимость кавычек, так как запись  $z := a$  могла бы иметь совсем другой смысл.)

С литерами связан еще один класс обозначений — обозначение строки (*row of character denotation*).

Обозначение строки отличается от литерного тем, что между кавычками помещается произвольное (но отличное от единицы) число литерных знаков и двойных кавычек в любой последовательности и любом сочетании.

Например:

$$""a \div + \div b \div "" \div \text{is} \div a \div \text{formula}"$$

( $\div$  символ пробела, т. е. символ, обладающий литерным значением, соответствующим пробелу при печати).

Значением обозначения строки объявляется одномерный массив литер с индексами от 1 до  $n$ , т. е. кратное значение, паспорт которого состоит из начала отсчета, равного 1, и одной пятерки (1,  $n$ , 1, 1, 1), где  $n$  — число компонент строки; элемент этого кратного значения с номером  $i$  ( $i = 1, \dots, n$ ) — литерное значение, которым обладает  $i$ -я слева компонента строки.

Может вызвать удивление особое выделение однокомпонентной строки: почему бы не считать литерное обозначение частным случаем обозначения строки? В [1] трактовка была именно такой, но впоследствии было замечено, что в силу некоторых тонкостей организации языка, о которых

сейчас мы еще не имеем возможности рассказать, особое выделение литерного обозначения дает дополнительные возможности, не умаляя старых. Суть дела заключается в том, что от простого значения перейти к кратному проще, чем наоборот. Поэтому литерное обозначение при необходимости с успехом исполняет роль строчного, но может использоваться и самостоятельно, что видно из приведенного выше присваивания.

Рассмотрим теперь *BITS denotation* (обозначение битов). Ясно, что здесь речь пойдет об изображении двоичных кодов, поэтому сразу же возникает вопрос, какими значениями будут обладать такие обозначения. Самое простое решение — сделать эти значения логическими массивами, т. е. кратными значениями вида *row of boolean*. Именно так было сделано в [1], где принято:

*BITS: row of boolean.*

С языковой точки зрения такое решение вполне удовлетворительно, но оно оказывается неудобным для эффективности реализации. Поэтому в [2] принята другая, более сложная трактовка вида битовых значений (см. § 12), и *BITS* определяется по-другому.

Однако так как в нашу задачу не входят заботы о нуждах реализации, мы здесь для простоты примем трактовку [1].

*row of boolean denotation: long symbol sequence option,*  
*flip flop sequence.*  
*flip flop: flip symbol; flop symbol.*

Поясним две синтаксические особенности первого правила, которые ранее еще не встречались.

Любое слово, заканчивающееся *option*, является понятием, имеющим два прямых порождения: пустое слово и слово, стоящее перед *option*. (Наличие *option*, таким образом, означает, что то, что перед ним стоит, «может быть, а может и не быть».) Поэтому в данном случае имеем:

*long symbol sequence option: EMPTY;*  
*long symbol sequence.*

(Вынесение пустого слова в метаязык связано с удобством чтения, дабы не было опасности не заметить пустого порождения.)

Далее, любое слово, оканчивающееся на *sequence*, также является понятием, порождения которого — соединения одного или более слов, стоящих перед *sequence* (*sequence* —

последовательность). Поэтому в данном случае будет:

*long symbol sequence: long symbol; long symbol,  
long symbol sequence.*

*flip flop sequence: flip flop; flip flop, flip flop sequence.*

Таким образом, синтаксически обозначение битов представляет собой последовательность «флипов» и «флопов» (*flip symbol* — 1, *flop symbol* — 0), перед которой может стоять один или более *long*. Например:

101

*long* 10100

Рассмотрим теперь семантику битового обозначения, т. е., каким значением оно обладает.

Для каждой длины, т. е. количества *long* в обозначении плюс единица, считается заданным, как и для чисел, свой диапазон, т. е. максимальное число битов (элементов значения). Диапазоны тоже фиксируются реализацией. Значение битового обозначения считается определенным только в том случае, когда число составляющих его «флип-флопов» не превышает диапазона, соответствующего количеству *long* в этом обозначении. В этом случае значением объявляется кратное значение, паспорт которого состоит из начала отсчета, равного 1, и одной пятерки (1, *n*, 1, 1, 1), где *n* равно д и а п а з о н у (а не числу флип-флопов). Все элементы этого значения с номерами, не превосходящими разности ( $n - m$ ), суть *false*. Элемент с каждым следующим *i*-м номером (т. е. номером, равным  $n - m + i$ ) — это *true*, если *i*-й флип-флоп есть 1, и *false*, если он 0.

Ясно, что этим моделируется ячейка с фиксированным числом разрядов (равным диапазону), в которой «нули добавляются слева». Набор стандартных операций над битовыми значениями, имеющийся в языке, моделирует обычные машинные команды, такие, например, как поразрядное логическое сложение и умножение, поразрядное сравнение, сдвиги и т. п., так что при желании можно перейти к работе на таком «квазимашинном» уровне. Однако с формальной точки зрения все это будет записано общими языковыми средствами.

Рассмотрим обозначения процедур, представляющие особый интерес в силу ранее упоминавшейся двойственности их использования. Прежде чем приводить соответствующие правила, охарактеризуем строение таких обозначений неформально.

Обозначение процедуры начинается с открывающей скобки и заканчивается закрывающей. Скобки нужны только для предотвращения двусмысленностей в тонких случаях использования обозначений процедур, и в силу расширений их часто разрешается опускать.

Содержимое скобок всегда начинается со списка формальных параметров, тоже заключенного в скобки (понятие обозначения процедуры без параметров отсутствует, так как его возможности перекрываются другими понятиями; как именно — увидим далее). Формальный параметр — это идентификатор, перед которым стоит описатель вида идентификатора. Такое вхождение идентификатора является определяющим. Формальные параметры могут отделяться друг от друга как запятой, так и точкой с запятой. При вызове рутины, которой обладает обозначение, некоторые действия, связанные с настройкой на фактические параметры, будут в зависимости от этого либо параллельными, либо последовательными.

Если обозначается функция какого-либо вида, то описатель этого вида помещается после списка формальных параметров, в противном случае этот описатель отсутствует. (Формально в этом случае стоит *void declarer*, который есть пустое слово.) Затем ставится разделитель — символ : (в представлении), после чего следует крупная синтаксическая единица — *унитарное выражение* нужного вида (в частности, «пустого», т. е. без вида). Любое выражение можно сделать унитарным (и даже первичным — частным случаем унитарного), заключив его в скобки, так что в обозначении процедуры может быть использован сколь угодно богатый набор языковых средств. Однако на уровне обозначения это всего лишь последовательность символов, из которой определенным образом формируется ее значение — рутинa. Только при вызове последней все это станет языковой конструкцией, подлежащей выполнению.

Приведем теперь синтаксические правила для обозначений процедур:

*procedure with PARAMETERS MOID denotation:*

*open symbol, open symbol, formal PARAMETERS,  
close symbol, MOID cast, close symbol.*

*formal PARAMETERS and PARAMETER:*

*formal PARAMETERS, comma, formal PARAMETER.*

*gomma*: *go on symbol*; *comma symbol*.

*formal MODE parameter*: *formal MODE declarer*,  
*MODE mode identifier*.

*MOID cast*: *virtual MOID declarer*, *cast of symbol*,  
*strongunitary MOID clause*.

Пока что мы еще ничего не знаем ни об описателях (*declarers*), ни об унитарных выражениях (*unitary clauses*). Об этом речь пойдет дальше (см. §§ 8 и 13). В остальном же приведенный синтаксис уже должен быть понятен (*go on symbol* « ; », *cast of symbol* « : »).

Заметим, что вся конструкция, стоящая после списка формальных параметров (*MOID cast*), сама является частным случаем унитарного выражения, и мы ее в свое время разберем.

Пример обозначения процедуры:

((*real a*, *real b*) *real* : *if a* > *b* *then b* *else a* *fi*)

(В АЛГОЛе-68 условные конструкции всегда завершаются символом *fi*, зеркальным по отношению к *if*.)

Заметим, что расширения позволяют в данном случае записать *real a*, *real b* в привычном виде *real a*, *b*.

Рассмотрим теперь, какими значениями обладают обозначения процедур. Эти значения формируются из обозначений следующим образом.

После каждого формального параметра помещается знак равенства, после которого следует специальный символ — *skip symbol* (*skip*). В нашем примере это приведет к такой записи:

(*real a* = *skip*, *real b* = *skip*)

Смысл этого преобразования состоит в том, что каждый формальный параметр превращается в синтаксическую единицу, называемую *описанием идентичности* (*identity declaration*). Описание идентичности — основное из описаний, имеющих в АЛГОЛе-68, с его помощью вводятся в рассмотрение все описываемые идентификаторы. Подробно мы рассмотрим его в § 10, сейчас же отметим только роль *skip*. В правой части описания идентичности должно стоять выражение нужного вида (здесь, в частности, вида *real*, т. е. арифметическое). Символ *skip* является в языке видовым «джокером»: выражение любого вида может в определенных позициях состоять из одного этого символа. При этом говорится, что такое выражение обладает «ка-

ким-то» значением нужного вида (каким — неизвестно, и нет средств это выяснить). Наличие такого символа бывает удобно, если в какой-то позиции синтаксически необходимо поместить какое-либо выражение, а его значение в данном случае безразлично.

В преобразовании формальных параметров в описания идентичности **skip**, таким образом, играет синтаксическую роль. При вызове рутины, как мы увидим в § 3, все **skip** будут заменяться фактическими параметрами (но фактическим параметром тоже может быть **skip**, если его значение для данного вызова несущественно).

Дальнейшее преобразование обозначения процедуры состоит в том, что убираются скобки, в которые были заключены формальные параметры, и после последнего формального параметра (точнее, после описания идентичности, в которое он превратился) ставится точка с запятой.

После этих преобразований обозначение процедуры превращается в одну из основных законченных синтаксических единиц — *закрытое выражение* (*closed clause*), синтаксис и семантику которого мы разберем в § 13. Эта последовательность символов и объявляется значением данного обозначения. Таким образом, рутины видов, начинающихся с *procedure with* (т. е. с параметрами), — это те последовательности символов, которые могут быть так получены.

Рассмотрим, например, следующее обозначение процедуры:

$((\text{int } a; \text{ real } x, \text{ real } y) \text{ real: if } a > 0 \text{ then } x \text{ else } y \text{ fi})$

Это обозначение обладает следующим значением (рутиной):

$(\text{int } a = \text{skip}; \text{ real } x = \text{skip}, \text{ real } y = \text{skip};$   
 $\text{real: if } a > 0 \text{ then } x \text{ else } y \text{ fi})$

(В силу расширений второе вхождение **real** как в обозначении, так и в записи рутины может быть опущено.) Разумеется, мы еще формально не знаем, что здесь написано с синтаксической и семантической точки зрения, но тут уж ничего не поделаешь, ибо по своему смыслу обозначения процедур могут содержать в себе все, чем располагает язык.

### § 3. Вызовы

Третий класс баз, который мы рассмотрим, связан с использованием рутин, а именно, с их вызовами. Эти базы порождаются *MOID call* и так и называются — *вызовы* (*calls*). Вызовы, порожденные *MODE call*, аналогичны указателям функций в АЛГОЛе-60, *void call* (*пустой вызов*) порождает аналоги оператора процедуры.

Мы рассмотрим все это вместе.

*MOID call: firm procedure with PARAMETERS MOID*  
*primary,*  
*open symbol, actual PARAMETERS, close symbol.*

Это — настоящий синтаксис вызова, в отличие от упрощенного, который мы приводили в качестве примера при рассмотрении метаязыка (см. § 2 гл. II). Подобно понятию обозначения процедуры без параметров и по той же причине в языке отсутствует понятие вызова без параметров.

Смысл слова *firm* (*твердый*) мы сейчас объяснять не будем, заметим только, что одним из порождений слова, в начале которого стоит *firm*, является слово, полученное удалением этого *firm*, так что пока можно на него не обращать внимания. Это относится также и к некоторым другим словам, которые нам встретятся. Наличие подобных слов связано с фундаментальной концепцией *приведения* видов и значений, которую мы рассмотрим специально в главе IV.

*Первичным выражением* (*primary*) может быть, в частности, любая база, а мы знаем уже три их класса. Приведем соответствующие примеры вызовов.

1)  $\sin(x + 1)$

Этот пример привычен — в роли первичного выражения выступает идентификатор.

2)  $((\text{int } a, b) \text{ int: if } a > b \text{ then } x \text{ else } y \text{ fi}) (x, y)$

Такая запись режет глаз тому, кто еще психологически не освоился с принципами АЛГОЛа-68, но формально так писать можно (нужно ли — это другой вопрос). Здесь в качестве первичного выражения выступает обозначение процедуры (формальные параметры записаны с использованием расширений).

3)  $a(b)(c)$

Этот пример тоже необычен. Так можно написать, если  $a$ ,  $b$  и  $c$  обладают, например, значениями следующих видов:

$a$  — *procedure with real parameter*  
*procedure with real parameter real*

(т. е. процедура с вещественным параметром, значением которой является процедура с вещественным параметром и вещественным значением),

$b - real$

$c - real$

Процедура, являющаяся значением вызова  $a(b)$ , вызывается с фактическим параметром  $c$ .

Забегая вперед, приведем пример с использованием четвертого класса баз:

$m[1, 2](x, y)$

Здесь  $m$  — двумерный массив процедур.

Упрощенно вызов выполняется следующим образом.

1. Выполняется его первичное выражение. После этого оно будет обладать некоторым значением — рутинной (рутиной потому, что вид этого значения согласно синтаксису вызова начинается с *procedure with*).

2. На место вызова помещается копия этой рутины, рассматриваемая отныне как закрытое выражение.

3. Все символы **skip** заменяются соответствующими фактическими параметрами.

4. Полученное таким образом закрытое выражение выполняется. Его значение, если оно имеется (т. е. в случае *MODE call*) и есть значение вызова.

5. Прежде чем начнется выполнение чего-нибудь другого, исходный вызов вновь ставится на свое место.

Четкая схема, не правда ли? Попробуем в качестве примера выполнить второй из вышеприведенных вызовов. Выполнение обозначения не влечет никаких действий. Согласно предыдущему оно обладает значением:

$(int\ a=skip, b=skip; int: if\ a > b\ then\ x\_else\ y\ fi)$

Такая рутина помещается на место вызова.

После замены всех **skip** фактическими параметрами получится такое закрытое выражение:

$(int\ a = x, b = y; int: if\ a > b\ then\ x\ else\ y\ fi)$

Формально мы еще не знаем, как такие конструкции выполнять, но смысл написанного ясен! Результирующим значением будет большее из значений  $x$  и  $y$ .

После завершения выполнения исходный вызов вновь встанет на свое место. (Разумеется, это не означает, что при

реализации действия будут именно такими, они могут быть любыми другими, дающими тот же эффект. Это касается всей семантики в целом.)

Отсюда видно, что в АЛГОЛе-68 принят единый механизм подстановки параметров, аналогичный подстановке «значением» в АЛГОЛе-60. Отсутствие аналога подстановки «наименованием» компенсируется наличием имен и рутин, которые при необходимости могут быть параметрами. (Например, если желательно достичь эффекта, эквивалентного подстановке наименованием арифметического выражения, то можно приписать соответствующему формальному параметру вид процедуры, вырабатывающей вещественное значение. Тогда при выполнении вызова с арифметическим выражением в качестве фактического параметра будет достигнут желаемый эффект, но почему это так, мы сейчас не имеем возможности рассказать, см. гл. IV.)

#### § 4. Вырезки

Последний класс баз — это *вырезки (slices)*. Вырезки являются далеко идущим обобщением переменных с индексами в АЛГОЛе-60. В то время как в АЛГОЛе-60 значения переменных с индексами могут быть только числами и логическими значениями, вырезки могут обладать значениями любого вида. Эти значения определенным образом формируются из некоторого исходного кратного значения и в известном смысле представляют собой его часть (отсюда и термин «вырезка»).

Мы ограничимся неформальным рассмотрением синтаксического строения вырезок (формальный синтаксис вырезок довольно сложен, и мы не будем здесь его приводить, оставляя читателю удовольствие встретиться с ним в [2] наедине).

Вырезка состоит из двух частей. Первая часть представляет собой первичное выражение, вид которого начинается либо с *row of* либо с *reference to row of*, т. е. первичное выражение, обладающее либо кратным значением, либо именем, ссылающимся на кратное значение. (В вызовах первичные выражения обладали процедурными значениями — рутинами, — и их вид согласно синтаксису начинался с *procedure with*. Правда, это имело место с точностью до непонятного слова *firm*, но и в правилах для вырезок есть подобные слова, которые мы пока что условились игнорировать.)

Вторая часть вырезки начинается с открывающей индексной скобки [ и заканчивается закрывающей индексной скобкой ]. (В строгом языке индексные скобки — это *sub symbol* и *bus symbol*.)

Между индексными скобками помещается одна или более конструкций двух типов: *индексов (subscripts)* и *триммеров (trimmers)*. Эти конструкции отделяются друг от друга запятой.

Индекс — это выражение вида *integral* (т. е. целое), не обязательно первичное, но и не вполне произвольное (дается максимум свободы, не приводящей к двусмысленностям).

Таким образом, в частности, вырезка может иметь привычное по АЛГОЛу-60 строение:

$x1 [i]$

$x2 [i, j]$

и т. п.

Триммер состоит из нескольких частей, некоторые из которых могут отсутствовать. Начинается он с выражения вида *integral*, называемого *жесткой нижней границей*. Затем следует разделитель *up to symbol* (: в представлении), за которым снова идет выражение вида *integral*. Это выражение называется *жесткой верхней границей*. Затем следует разделитель *at symbol* (at) и снова выражение вида *integral*, называемое *новой нижней границей*.

Возможности для всех границ такие же, как и для индексов. Таким образом, полный триммер может выглядеть, например, так:

$2:n \text{ at } 0$

Но из всех этих частей разрешается опускать все, кроме :. Поэтому триммеры могут иметь следующее строение:

$2:n \quad 2: \text{at } 0$

$2: \quad : \text{at } 0$

$: n \quad :$

(Единственное ограничение — если написано *at*, то новая нижняя граница необходима.)

Наконец, в силу расширений триммер : может быть заменен пустым словом (т. е. в этом случае можно опустить и двоеточие.)

## Пример вырезки:

$A [i, 1:10 \text{ at } 5, j, : \text{ at } 0, 5, : n]$

Число индексов и триммеров в вырезке не обязательно должно быть равно числу *row of* в виде первичного выражения перед скобками, оно должно быть только не больше его.

Рассмотрим теперь семантику вырезок. Сначала мы приведем ее почти в таком виде, как в [2], устранив и изменив только некоторые детали, потом дадим пояснения.

Вырезка выполняется следующим образом:

Шаг 1. Ее первичное выражение (перед скобками) и все ее индексы, нижние, верхние и новые границы выполняются параллельно (т. е. выполняется параллельное действие, компонентами которого являются выполнения этих частей вырезки).

Шаг 2. Рассматривается кратное значение, которое есть или на которое ссылается значение первичного выражения, полученное на первом шаге; делается копия паспорта этого значения, и в ней все состояния полагаются равными 1.

Шаг 3. Рассматривается «триндекс» (триммер или индекс), следующий за открывающей скобкой, заводится «указатель»  $i$ , который полагается равным 1.

Шаг 4. Если рассматриваемый триндекс есть  $:$ , то шаг 6, иначе, если он триммер, то шаг 5, в противном случае (т. е. в случае индекса) пусть  $k$  — его значение (полученное на первом шаге); если  $l_i \leq k \leq u_i$  ( $i$  — указатель;  $l_i$ ,  $u_i$  — границы из  $i$ -й пятерки копии паспорта, заведенной на втором шаге), то начало отсчета в копии паспорта увеличивается на  $(k - l_i) \times d_i$ ,  $i$ -я пятерка «помечается», и шаг 6; в противном случае дальнейшее выполнение не определено.

Шаг 5. Вычисляются величины  $l$ ,  $u$  и  $l'$  следующим образом: если рассматриваемый триммер содержит жесткую нижнюю (жесткую верхнюю) границу, то  $l$  ( $u$ ) есть ее значение, в противном случае  $l$  ( $u$ ) есть  $l_i$  ( $u_i$ );

если имеется новая нижняя граница, то  $l'$  — ее значение, в противном случае  $l' = l$ ;

если теперь  $l_i \leq l$  и  $u \leq u_i$ , то начало отсчета в копии паспорта увеличивается на  $(l - l_i) \times d_i$ , после чего  $l_i$  заменяется на  $l'$ , а  $u_i$  заменяется на  $(l' - l) + u$ ; в противном случае дальнейшее выполнение не определено.

Шаг 6. Если за рассматриваемым триндексом следует запятая, то вместо него рассматривается следующий за ней триндекс, указатель увеличивается на 1, и шаг 4; в противном случае из копии паспорта удаляются все пятерки, помеченные в четвертом шаге, и шаг 7.

Шаг 7. Если преобразованная копия паспорта теперь содержит хотя бы одну пятерку, то формируется кратное значение, состоящее из этой копии — паспорта формируемого значения — и тех элементов рассматриваемого кратного значения (см. шаг 2), которые этот паспорт описывает, это значение рассматривается вместо полученного на втором шаге; в противном случае (т. е. в случае отсутствия пятерок в копии паспорта) рассматривается тот элемент исходного кратного значения, номер которого равен началу отсчета в копии паспорта.

Шаг 8. Если значение первичного выражения (полученное на первом шаге) есть имя, то значение вырезки есть имя, ссылающееся на рассматриваемое значение; в противном случае значение вырезки — само рассматриваемое значение.

Перед нами — самая настоящая программа с разветвлениями и условными переходами. Рекомендуем, прежде чем переходить к чтению комментариев, попробовать «проиграть» эту программу на каком-нибудь простом примере.

Прокомментируем теперь выполнение вырезки.

Первый шаг интересен только тем, что он представляет собой параллельное действие. Это делает неопределенным смысл такой, например, записи:

$$m [(a := i), (a := a + 1)]$$

(в АЛГОЛе-68 присваивание может входить в состав выражений, далее мы это разберем), так как семантика языка не определяет последовательность присваиваний.

Значение первичного выражения есть либо кратное значение, либо имя, ссылающееся на кратное значение, в зависимости от того, начинается ли его вид с *row of* или с *reference to row of*.

На втором шаге заготавливается паспорт будущего кратного значения и все состояния  $s_i$  и  $t_i$  этого паспорта полагаются равными 1. Этим запрещается менять границы у подмассивов (при выполнении присваивания, осуществляющего смену границ, сработает соответствующая блокировка). Запрещение естественно: ведь рассматриваются

только прямоугольные массивы, а осуществление, скажем, присваивания одной из четырехкомпонентных строк матрицы значения пятикомпонентного вектора должно было бы привести к появлению незаконного объекта — непрямоугольного массива.

Роль указателя  $i$ , заводимого на третьем шаге, ясна из динамики выполнения вырезки — это просто порядковый номер текущего индекса или триммера.

Рассмотрим четвертый шаг. Первое «если» в нем обязательно, так как  $i$  частный случай триммера. Просто в этом случае заведомо не нужно выполнять действий пятого шага, и дополнительный анализ представляет собой, так сказать, «оптимизацию программы по времени».

Второе «если» существенно, оно отделяет индекс от триммера, имеющего свою семантику. Трактовку индекса дает дальнейшая часть четвертого шага, трактовку триммера — параллельный пятый шаг. Как в том, так и в другом случае производятся преобразования копии паспорта, к которым формально сводится формирование подмассива. Эти преобразования приводят к тому, что полученный паспорт описывает не все элементы исходного кратного значения, а только некоторые из них. Эти элементы и образуют подмассив. Таким образом, подмассив получается удалением части элементов массива путем преобразований копии паспорта, делающих такие элементы «посторонними». (Поэтому множество номеров элементов подмассива, как правило, не является отрезком ряда целых чисел, что соответствует замечанию в конце § 3 гл. I).

В случае индекса (т. е. в четвертом шаге) преобразование паспорта представляет собой увеличение начала отсчета на

$$(k - l_i) \times d_i$$

и «пометку»  $i$ -й пятерки ( $k$  — значение индекса,  $i$  — его порядковый номер). (Смысл предварительной проверки неравенства  $l_i \leq k \leq u_i$  ясен — это блокировка выхода значения индекса за пределы границ.)

Пометка пятерки производится для того, чтобы удалить эту пятерку из паспорта — в конце шестого шага из паспорта удаляются все помеченные пятерки. (Можно было бы удалить пятерку из паспорта сразу, т. е. в четвертом шаге, но тогда здесь же нужно было бы дополнительно уменьшить на 1 указатель  $i$ , так как следующая пятерка стала бы уже не  $(i + 1)$ -й, а  $i$ -й.)

Вспомним (см. § 3 гл. I), что номер любого элемента кратного значения может быть представлен в такой форме:

$$I(r_1, \dots, r_n) = (r_1 - l_1) \times d_1 + \dots + (r_i - l_i) \times d_i + \dots + (r_n - l_n) \times d_n + c \quad (*)$$

$$l_i \leq r_i \leq u_i; \quad i = 1, \dots, n;$$

где  $n$  — число пятерок паспорта.

Следовательно, удаление из паспорта  $i$ -й пятерки удаляет  $i$ -е слагаемое из (\*), а увеличение начала отсчета как бы ставит на его место фиксированное число  $(k - l_i) \times d_i$ . Это означает, что из массива не выбрасываются только те элементы, для которых в (\*) будет  $r_i = k$ .

Таким образом,  $i$ -я позиция индекса в вырезке фиксирует множество тех элементов массива, у номеров которых в записи (\*)  $r_i$  равны значению этого индекса, а также уменьшает на 1 размерность полученного таким образом подмассива.

Ясно, что если все индексные позиции в вырезке занимают индексы и их столько же, сколько пятерок в паспорте исходного кратного значения, то из паспорта будут удалены все пятерки, а сформированное при этом начало отсчета будет номером элемента, соответствующего данному набору индексов, что дает прямой аналог переменной с индексами в АЛГОЛе-60.

Пусть, например, паспорт кратного значения имеет начало отсчета, равное 1, и две пятерки: (1, 2, 2, 1, 1) и (1, 2, 1, 1, 1), и этим значением обладает идентификатор  $m$  (например,  $m$  имеет описатель [1 : 2, 1 : 2] real).

Рассмотрим вырезку  $m$  [2, 2].

При первом выполнении четвертого шага начало отсчета увеличится на  $(2-1) \times 2 = 2$  и станет равным 3, т. е. выбросится вся первая строка матрицы  $m$ . При втором его выполнении произойдет увеличение начала отсчета на  $(2-1) \times 1 = 1$ , и оно станет равным 4, а это и есть номер нужного элемента.

Однако индексы могут еще и чередоваться с триммерами, трактовку которых, как уже говорилось, дает пятый шаг выполнения вырезки. Рассмотрим этот шаг.

Предположим сначала, что новая нижняя граница в триммере отсутствует. Тогда величина  $l'$  в пятом шаге не нужна, так как в этом случае  $l' = l$  (и поэтому  $l_i$  будет заменяться на  $l$ , а  $u_i$  на  $u$ ). Пусть далее отсутствует жесткая нижняя граница. Тогда не нужна и  $l$ , так как будет  $l =$

$= l_i$ , начало отсчета не изменится, и только  $u_i$  заменится на  $u$ . Наконец, если отсутствует и жесткая верхняя граница, то не нужна и  $u$ , так как тогда  $u = u_i$ , и в паспорте ничего не изменится (поэтому в случае ! делается обход пятого шага). Таким образом, не написать какую-либо из границ в триммере — это все равно, что написать прежнюю границу \*). (Если проследить дальнейшее выполнение вырезки, то легко видеть, что, например, вырезка  $v [ \ ]$  с точностью до состояний обладает таким же значением, как и идентификатор  $v$ , если  $v$  — идентификатор одномерного массива.)

Пусть теперь в триммере присутствует только жесткая верхняя граница. Тогда в паспорте  $u_i$  заменяется ее значением  $u$  (при условии  $u \leq u_i$ , иначе паспорт начал бы описывать неизвестно что). Это означает, что из массива выбрасываются все элементы, к которым можно было бы «подступиться» помещением индекса, значение которого больше  $u$  (но не превосходит  $u_i$ ) в  $i$ -ю индексную позицию.

Например, в состав значения вырезки  $v [ : 3]$ , где идентификатор  $v$  имеет описатель  $[1 : 5]$  real, будут входить только первые три элемента значения  $v$ , а в состав значения вырезки  $m [2, : 3]$ , где идентификатор  $m$  имеет описатель  $[1 : 4, 1 : 4]$  real, будут входить первые три элемента второй строки матрицы  $m$ .

Комбинируя триммеры и индексы, можно получать всевозможные прямоугольные вырезки. Переставим, например, позиции в предыдущем примере, т. е. рассмотрим вырезку  $m [ : 3, 2]$ . Предоставляем убедиться, что в состав элементов ее значения будут входить первые три элемента второго столбца матрицы  $m$ . (Здесь триммер : 3 выбрасывает четвертую строку, а индекс 2 — все элементы не второго столбца оставшихся строк, т. е. 1-й, 3-й и 4-й их элементы.)

Наличие в триммере только жесткой нижней границы играет аналогичную роль, но в этом случае требуется изменить начало отсчета. Действительно, рассмотрим вырезку  $u [3 : ]$ . Пусть паспорт значения  $u$  имеет начало отсчета,

---

\*) В [2], в отличие от [1], принята несколько иная трактовка новой нижней границы: считается, что не написать ее — это все равно, что написать 1. Такая автоматическая установка нижних границ на 1 (на 1 потому, что считается, что наиболее естественно начинать индексацию именно от 1) дает в ряде случаев возможность упрощения записи, однако в других случаях требует большей осторожности при использовании вырезок, потому что эта установка происходит только при выполнении триммеров. Так как для наших целей такие детали не существенны, мы для простоты и единообразия приняли здесь трактовку [1].

равное 1, и пятерку (1, 5, 1, 1, 1). Мы хотим, чтобы в состав значения вырезки входили только третий, четвертый и пятый элементы значения  $v$ . Но если мы просто изменим пятерку на (3, 5, 1, 1, 1), то не достигнем цели, так как такой паспорт будет описывать элементы со следующими номерами:

$$1 + (3 - 3) = 1, \quad 1 + (4 - 3) = 2, \quad 1 + (5 - 3) = 3,$$

т. е. первый, второй и третий.

Дело в том, что вычитается здесь уже не 1, а измененная граница, а именно 3. Поэтому соответствующим образом увеличивается и начало отсчета. В данном случае оно становится равным  $1 + (3 - 1) \times 1 = 3$ , и мы получаем, что описываются элементы с нужными номерами:

$$3 + (3 - 3) = 3, \quad 3 + (4 - 3) = 4, \quad 3 + (5 - 3) = 5.$$

Выясним теперь роль новой нижней границы. Пусть в триммере присутствует только она одна, и  $l'$  — ее значение. Тогда  $l_i$  заменится на  $l'$ , а  $u_i$  заменится на  $(l' - l_i) + u_i$  (так как в нашем случае  $l = l_i, u = u_i$ ). От этого множество описываемых паспортом элементов не изменится, но зато теперь, чтобы «подступиться» к тому элементу, для которого раньше в  $i$ -й позиции нужно было ставить  $k$ , теперь нужно ставить  $k + (l' - l_i)$ , т. е. просто изменяется внешняя индексация элементов.

Например, следующая вырезка:

$$v [: \text{at } 0] [3]$$

обладает таким же значением, как и  $v [4]$  (надеемся, что такая «двойная» вырезка уже не вызывает недоумения).

Присутствие в триммере всех или нескольких границ дает наложение рассмотренных эффектов, которое в пятом шаге выполнения вырезки уложено в единую компактную схему, сначала, впрочем, могущую показаться не очень понятной.

Рассмотрим теперь шестой и седьмой шаги. Начало шестого шага — это заикливание программы на рассмотрение следующего триммера или индекса, если таковой имеется. Конец шестого шага и седьмой шаг подчеркивает разные роли индексов и триммеров: каждый индекс уменьшает на единицу размерность результата, триммер же не меняет размерности. Поэтому, например, значение  $v [1]$  — не кратное, в то время как значение  $v [1 : 1]$  — кратное, в состав которого входит один элемент.

Наконец, восьмой шаг очевиден — значением вырезки объявляется либо само сформированное значение, либо имя, на него ссылающееся, в зависимости от вида первичного выражения перед скобками. (Считается, что если есть имя, ссылающееся на кратное значение, то имеются и имена, ссылающиеся на все подзначения, которые можно из него вырезать.)

Заметим, что из программы выполнения вырезов следует, что если индексных позиций в вырезке меньше, чем размерность массива, то это будет трактоваться так, как если бы каждую недостающую позицию занимал триммер : Единственную выгоду, которую при этом получает программист — это разрешение не писать такие триммеры в заключительных позициях (а так как в силу расширений двоеточия можно опускать, то это сводится к разрешению не писать «заключительные запятые»). В других же случаях эти триммеры (запятые) не могут быть опущены. Нам, однако, представляется, что это — слишком малая выгода по сравнению с опасностью просто ошибиться и пропустить позицию (такая возможность весьма существенна). Здесь же вместо сигнализации об ошибке будет выполняться нечто совсем далекое от замысла программиста.

## § 5. Выделения полей

В настоящем параграфе мы рассмотрим конструкции, связанные с полями структурных значений, аналогично тому, как вырезки были связаны с элементами и «подзначениями» кратных значений. Но по сравнению с вырезками средства работы со структурными значениями в этом смысле гораздо беднее и сводятся действительно только к *выделению поля*, т. е. к рассмотрению значения, являющегося полем структуры (структурного значения). (Но не следует забывать, что это значение само может быть структурным, именем, ссылающимся на структуру и т. д.)

*Выделение поля (selection)* — это также по смыслу одна из базовых конструкций и формально не считается базой только из синтаксических соображений; далее мы приведем пример, в котором включение выделений поля в состав баз вызвало бы неоднозначность трактовки.

Мы приведем синтаксис выделений полей несколько упрощенно. Дело в том, что в [2] названия полей структур (синтаксически это идентификаторы) входят в состав их видов. Это вызвано отнюдь не видовыми соображениями

(в самом деле, не все ли равно, как назвать поле), а только грамматическими: появляется возможность синтаксического запрета появления посторонних идентификаторов в роли названий полей (в правилах синтаксиса фигурируют виды \*). Мы же определили ранее *MODE*, игнорируя это обстоятельство. Поэтому и синтаксис выделения полей мы определим так же:

*REFETY MODE selection: MODE field selector, of symbol,  
weak REFETY structured  
with LFIELDSETY  
MODE field RFIELDSETY  
secondary.*

*REFETY: reference to; EMPTY.  
LFIELDSETY: FIELDS and; EMPTY.  
RFIELDSETY: and FIELDS; EMPTY.*

*Выделитель поля (field selector)* — это и есть идентификатор, играющий роль названия поля. Он появляется в описателях структурных значений, и, таким образом, каждому полю оказывается поставленным в соответствие свой выделитель, т. е. подобно тому, как в случае кратных значений рассматривались пары (*элемент, номер*), здесь рассматриваются пары (*поле, выделитель*). В выделениях полей могут встречаться только выделители, соответствующие какому-либо полю данного структурного значения (как уже говорилось, это достигается включением выделителя в вид структур).

После выделителя поля следует разделитель *of (of symbol)*, за которым идет выражение, вид которого начинается либо с *structured with*, либо с *reference to structured*

---

\*) С этим утверждением можно и не согласиться. Действительно, вместе с каждым структурным видом необходимо ввести и операции, позволяющие выделить из структурного значения данного вида каждое из значений его полей. Названия полей как раз и обслуживают эту потребность. Кроме того, названия полей используются в АЛГОЛе-68 для различения структурных видов, у которых виды соответствующих полей совпадают. Например, виды

*struct (real re, im)*

и

*struct (real mod, arg)*

считаются различными лишь потому, что полям даны различные названия. (Это одно из слабых мест языка — логичнее было бы использовать для различения таких структурных видов названия — индиканты видов, которые можно давать самим видам, см. § 12 гл. IV.) (*Прим. ред.*)

*with*, т. е. обладающее либо структурным значением, либо именем, ссылающимся на структурное значение (аналогично виду первичного выражения в вырезке по отношению к *row of*). Выражение, стоящее после *of*, относится к классу *вторичных* (*secondary*). Вторичным может быть любое первичное выражение (в частности, любая база), а также выделение поля и конструкция еще одного класса (*генератор*), который мы рассмотрим в § 9.

Слово *weak* (слабый) связано с приведениями.

Примеры выделения полей:

*age of pl 1*

*father of father of algol*

Рассмотрим теперь такую конструкцию:

*a of b [i]*

Если бы выделение поля было одной из баз, то, не зная вида *b*, можно было бы дать две трактовки этой конструкции:

*(a of b) [i]*

тогда значение *b* — структура, а ее поле *a* — массив;

*a of (b [i])*

тогда значение *b* — массив структур.

Исключение выделения поля из баз (и даже из первичных выражений) делает возможным только одну трактовку, а именно, последнюю, так как в вырезке перед индексными скобками обязано стоять первичное выражение. Это облегчает синтаксический анализ, ибо нет нужды справляться о виде *b*. Если все же нужна именно первая трактовка, то круглые скобки необходимы.

Семантика выделения поля достаточно проста:

1) выполняется вторичное выражение, стоящее после *of*; рассматривается структурное значение, которое есть или на которое ссылается значение этого выражения;

2) если значение выражения — имя, то значением выделения поля является имя, ссылающееся на то поле рассматриваемого структурного значения, которому поставлен в соответствие выделитель, стоящий перед *of*, в противном случае значение выделения поля — само это поле.

После детального рассмотрения разветвленной семантики вырезок это выполнение должно быть понятно без пояснений. (Здесь тоже считается, что если есть имя, ссылающееся на структуру, то есть и имена, ссылающиеся на ее поля).

## § 6. Присваивания

Как в АЛГОЛе-60, основное средство манипуляций со значениями в АЛГОЛе-68 — *присваивание* (*assignment*). До сих пор это понятие употреблялось неформально в расчете на ассоциации читателя, связанные с АЛГОЛом-60, теперь же мы достаточно подготовлены к его специальному рассмотрению.

Присваивание в АЛГОЛе-68 — это частный случай выражения (так что оно обладает определенным значением), но оно может занимать и позицию оператора (тогда его значение игнорируется, учитывается только эффект выполнения).

Вообще в АЛГОЛе-68 нет жесткого различия между операторами и выражениями — есть просто конструкции, подлежащие выполнению. После выполнения они либо обладают значениями, либо нет. Если значение есть, а занимаемая позиция такова, что оно не нужно (позиция оператора), то оно «забывается» (конечно, если значение нужно, а его нет, то это ошибка, и синтаксис следит за недопустимостью такой ситуации).

Понятия *оператор* (*statement*) и *выражение* (*expression*) появляются только во вспомогательных правилах со звездочками, т. е. в помощь читателю. Выражения порождаются с помощью *MODE clause*, операторы — с помощью *void clause*, в синтаксисе же чаще всего фигурирует просто *MOID clause*. Мы поэтому использовали и будем использовать один термин для всех *clause* — *выражение*.

Синтаксис присваивания следующий:

*reference to MODE assignment: reference to MODE destination, becomes symbol, MODE source.*  
*reference to MODE destination: soft reference to MODE tertiary.*

*MODE source: strong unitary MODE clause.*

Таким образом, присваивание состоит из двух частей разделенных символом  $:=$  (представлением *becomes symbol*).

Слева от знака присваивания стоит его *назначение* (*destination*). Назначение — это выражение, относящееся к классу *третичных* (*tertiary*). В частности, назначение может быть и первичным и вторичным выражением, но вот присваиванием быть не может, так как последнее к третичным не относится.

Вид назначения (равно как и самого присваивания) должен начинаться с *reference to*, т. е. оно должно обладать именем. Поэтому всюду, где в правилах фигурирует *MODE assignation*, возможен тупик, так как присваивание определено только для имен.

Справа от знака присваивания стоит его *источник* (*source*) — выражение того вида, который следует после *reference to* в виде назначения. Источник относится к обширному классу *унитарных* (*unitary*) выражений. Этот класс включает в себя и третичные выражения, и сами присваивания (в § 13 мы приведем всю синтаксическую иерархию унитарных выражений). В частности, источником снова может быть присваивание, так что мы получаем возможность многократного присваивания, привычного по АЛГОЛу-60, без усложнения синтаксиса.

Слова *soft* (*мягкий*) и *strong* (*сильный*) связаны с приведениями.

Примеры присваиваний:

$$\begin{aligned} x &:= 3.14 \\ \text{if } a > b \text{ then } x \text{ else } y \text{ fi} &:= 1 \\ x &:= y := 0 \end{aligned}$$

Заметим, что многократное присваивание должно было бы вызвать затруднения у внимательного читателя. Действительно, пусть в последнем примере и  $x$  и  $y$  — вида *reference to integral*.

Тогда мы имеем дело с *reference to integral assignation*, и поэтому его источником должно быть выражение вида *integral*. Но в нашем примере источником является присваивание  $y := 0$ , а оно имеет вид *reference to integral*. Вот здесь-то, в частности, и сработает аппарат приведений (в данном случае благодаря слову *strong* в синтаксисе источника), который при выполнении источника автоматически заставит перейти от имени к целому значению, на которое это имя ссылается (но присваивание  $y := 0$  будет перед этим выполнено).

Но, как уже говорилось, механизм приведений мы будем рассматривать специально.

Обратимся теперь к семантике присваивания. В АЛГОЛе-60 семантика оператора присваивания буквально следующая: «Оператор присваивания служит для присваивания значений переменным». Коротко и, строго

говоря, неясно. Дело спасает простота ситуации, ведь присваиваться могут только числа и логические значения.

Но АЛГОЛ-68 имеет дело с большим и разнообразным хозяйством: здесь и кратные значения, и структурные, и рутины и т. д. Все это нужно учесть, предусмотреть возможные неприятности и уложить в единую схему. Поэтому семантика присваивания подобно семантике вырезок тоже представляет собой многошаговую программу. Как и в случае вырезок, мы сначала приведем ее почти в таком виде, как в [2] (устранив только некоторые мелкие детали), после чего дадим соответствующие комментарии.

I. Присваивание выполняется следующим образом:

Шаг 1. Его источник и назначение выполняются параллельно.

Шаг 2. Значение источника *присваивается* (см. II) имени, являющемуся значением назначения.

Шаг 3. Значением присваивания является значение его назначения (т. е. имя).

II. Значение присваивается имени следующим образом:

Шаг. 1. Если область действия имени не больше области действия присваиваемого значения, то шаг 2, в противном случае дальнейшее выполнение не определено.

Шаг 2. Рассматривается значение, на которое ссылается имя; если вид имени начинается с *reference to structured with* или с *reference to row of*, то шаг 3, в противном случае рассматриваемое значение *замещается* (см. III) копией присваиваемого значения, и присваивание завершено.

Шаг 3 \*). Если рассматриваемое значение — структурное, то шаг 5; в противном случае пусть  $(l_i, u_i, d_i, s_i, t_i)$ ,  $i = 1, \dots, n$  — все пятерки его паспорта, тогда:

а) для всех  $i$  от 1 до  $n$ : если  $s_i = 0$  ( $t_i = 0$ ), то  $l_i(u_i)$  полагается равной  $i$ -й нижней ( $i$ -й верхней) границе в паспорте присваиваемого значения;

б) для  $i = n, n-1, \dots, 2$  шаг  $d_{i-1}$  полагается равным  $(u_i - l_i) \times d_i$ ;

---

\*) В процессе работы над языком, уже после опубликования [2], выполнение этого шага было изменено, так как обнаружили нежелательные эффекты в некоторых тонких случаях. Однако это изменение приводит, на наш взгляд, к еще большим неприятностям, причем отнюдь не в тонких случаях. Поэтому здесь мы оставляем трактовку [2].

в) если хотя бы одно из состояний  $s_i$  или  $t_i$  равно 0, то паспорт рассматриваемого значения, модифицированный, как указано выше, становится паспортом некоторого кратного значения (какого, неважно) того же вида, имя начинает ссылаться на это значение, и оно начинает рассматриваться вместо рассматриваемого прежде.

Шаг 4. Если для всех  $i$  от 1 до  $n$  границы  $l_i(u_i)$  в паспорте рассматриваемого значения, возможно модифицированного на шаге 3, равны границам  $l_i(u_i)$  в паспорте присваиваемого значения, то шаг 5; в противном случае дальнейшее выполнение не определено.

Шаг 5. Каждое поле (каждый элемент) присваиваемого значения присваивается (в каком порядке, не определено) имени, ссылающемуся на соответствующее поле (соответствующий элемент) рассматриваемого значения, и присваивание завершено.

III. Данное значение замещается другим значением следующим образом:

1) имя, которое ссылается на данное значение, начинает ссылаться на другое значение (и, следовательно, перестает ссылаться на старое значение);

2) каждое имя, которое ссылается на структурное или кратное значение, компонентой которого является данное значение, начинает ссылаться на структурное или кратное значение, образуемое путем замены этой компоненты другим (замещающим) значением. (Компоненты структур — их поля, компоненты кратных значений — их подзначения, т. е. значения вырезок, в частности, их элементы. Компоненты компонент тоже считаются компонентами исходного значения.)

Вот оказывается, во что можно превратить обыкновенное присваивание, если взяться за дело всерьез! Попытаемся теперь разобраться, что же здесь происходит.

Рассмотрим сначала выполнение всего присваивания. Первый шаг, представляющий собой параллельное действие, не должен уже нуждаться в комментариях. Второй шаг содержательно тоже ясен, а что формально значит «присвоить», будет разобрано дальше. Третий же шаг может вызвать удивление: почему значением присваивания объявляется имя, которому что-то присваивалось, а не то значение, которое этому имени присваивалось? (Формально, конечно, так и должно быть, так как согласно синтаксису вид присваивания — вид имени, но вопрос ставится содержательно: а почему сделано именно так?) В [1] была

принята как раз другая трактовка, но в процессе «доводки» языка выяснилось, что при необходимости можно легко перейти от имени к значению, на которое это имя ссылается: за этим следит аппарат приведений (как, например, в вышеприведенном многократном присваивании). Так что здесь по сравнению с [1] просто дается больше возможностей (обратного пути — от значения к имени — нет).

Теперь рассмотрим, что значит присвоить значение имени. Первый шаг был уже прокомментирован в главе I при рассмотрении областей действий \*). Правда, мы еще не знаем, какие конкретные области действия бывают у имен, но об этом речь впереди.

Второй шаг начинается с анализа, не является ли присваиваемое значение кратным или структурным, для которых предусмотрены свои ветки. Правда, анализируется вид имени, которому присваивается это значение, но синтаксис обеспечивает согласование обоих видов. В случае отрицательного результата этого анализа (а к этому случаю, как мы увидим, все в конце концов сводится) присваивание сразу же завершается путем замещения.

Что это значит? В общем случае просто то, что стрелка, ведущая от имени к значению, на которое оно до сих пор ссылалось, начинает указывать на присваиваемое значение (ведь нам же нужно менять направления стрелок!). Но может оказаться, что старое значение (т. е. то, на которое имя ссылалось) входит в состав другого значения (кратного или структурного), т. е. является его компонентой (что такое компоненты структурных и кратных значений, сказано в семантике замещения). На это объемлющее значение тоже могло ссылаться некоторое имя. В таком случае считается, что замещение определенным образом влияет и на это имя. А именно, в объемлющем значении старое значение заменяется новым (т. е. тем, на которое начало ссылаться первое имя) и это другое имя начинает

---

\*) В [2] предусмотрена еще одна проверка на первом шаге: если присваиваемое значение — это имя, ссылающееся на компоненту (элемент или подмассив) кратного значения с хотя бы одной подвижной границей, то дальнейшее выполнение не определено. Смысл этого ограничения ясен — номер элемента или паспорт подмассива, на который ссылается присваиваемое имя, определяются по текущим значениям границ кратного значения. Если впоследствии эти границы получают новые значения, то названные выше характеристики компоненты кратного значения, а вместе с ними и имя этой компоненты потеряют смысл (напомним, что имя является аналогом адреса значения в памяти).

ссылаться на значение, полученное таким способом. В свою очередь объемлющее значение тоже может быть компонентой другого значения, и на это значение тоже может ссылаться имя. Тогда замещение точно так же влияет и на это имя и т. д.

Анализ в начале третьего шага отделяет случай кратного значения от случая структурного, в котором выполнение продолжается сразу с пятого шага. Дальнейшие действия третий шаг производит уже только над кратными значениями. Здесь-то и выясняется роль состояний  $s_i$  и  $t_i$  в паспорте старого значения: их нулевые значения разрешают смену соответствующих границ. Это делается так: в случае наличия хотя бы одного нулевого состояния формируется вспомогательное кратное значение, имеющее паспортом старый паспорт со всеми измененными границами. Элементы этого значения могут быть произвольными, так как они все равно заместятся после выполнения присваивания. Вспомогательное значение нужно четвертому шагу только в качестве «рассматриваемого» значения, на которое ссылается данное имя. В случае отсутствия нулевых состояний это вспомогательное значение не заводится, так как в старом паспорте ничего не меняется, и в роли рассматриваемого значения может выступить старое.

Заметим, что в случае смены границ производится и пересчет шагов  $d_i$ , соответствующий новому числу элементов в каждом измерении (шаг  $d_n$  при этом остается прежним).

Четвертый шаг проверяет совпадение всех границ рассматриваемого (т. е. старого или вспомогательного) и присваиваемого значений. Несовпадение какой-либо границы означает, что соответствующее состояние в старом паспорте равно 1 (иначе на место этой границы в силу третьего шага помещалась бы новая граница, разумеется, совпадающая сама с собой), т. е. эту границу менять запрещено, в то время как у присваиваемого значения граница другая. В этом случае дальнейшее выполнение объявляется неопределенным. (Например, пусть что-то присваивается имени вырезки. Как мы видели при рассмотрении вырезок, значение, на которое это имя ссылается, имеет все состояния, равные 1. Поэтому любая попытка изменить присваиванием границы подмассива будет в этом месте блокирована.)

Наконец, на пятом шаге «структурная» и «кратная» ветки соединяются. Здесь все элементы или поля присваиваемого значения присваиваются именам, ссылающимся на

соответствующие элементы или поля старого значения (как уже говорилось, считается, что все такие имена существуют).

Таким образом, программа выполнения присваивания рекурсивна, она вызывает сама себя, и весь этот процесс разворачивается дальше: если элемент или поле — не массив и не структура, то дело заканчивается замещением на втором шаге, в противном случае будет увеличиваться глубина рекурсии на пятом шаге и т. д.

(Может вызвать удивление, что на пятом шаге говорится только о присваивании элементов или полей и не упоминается о том, что должно что-то произойти и с именем, ссылающимся на само присваиваемое кратное или структурное значение. Но этого и не нужно особо оговаривать в силу семантики замещения.)

Как мы видим, на деле все оказывается довольно естественным, громоздкость и неудобочитаемость связаны с действительной необходимостью все предусмотреть. В обычных же случаях, например, при присваивании типа  $x := 1$  дело сводится просто к «перекидке» стрелки, что, очевидно, моделирует все возможности оператора присваивания в АЛГОЛе-60.

## § 7. Ядра

Конструкции, название которых приблизительно можно перевести как *ядро* (*cast*), уже встречались при рассмотрении обозначений процедур. Они тоже являются частным случаем унитарных выражений и могут соответствующим образом использоваться. Напомним синтаксис ядер:

*MOID cast: virtual MOID declarer,*  
*cast of symbol, strong unitary MOID clause.*

Таким образом, любое ядро состоит из описателя (в случае *void* он будет пустым), разделителя *cast of symbol* ( $:$ ) и унитарного выражения того же вида, что и описатель. Слово *strong*, как уже говорилось, связано с приведениями.

Семантика ядра проста: его выполнение состоит из выполнения выражения, входящего в его состав.

Зачем же понадобилась такая синтаксическая единица, как ядро, если его выполнение состоит лишь из выполнения его составной части, почему просто не писать всегда эту составную часть? Пока мы еще не можем исчерпывающе

ответить на этот вопрос, так как ответ связан с аппаратом приведенных. В общих чертах дело заключается в следующем.

Язык устроен так, что одинаковые последовательности символов могут трактоваться как выражения разных видов в зависимости от того, какую позицию они занимают, причем трактовка каждой позиции известна, ее-то и осуществляет аппарат приведенных. Но может оказаться желательным вмешаться в работу этого аппарата и заставить приписать выражению тот вид, который нужен (в рамках возможностей, конечно). Это можно сделать, превратив выражение в ядро, т. е. поставив перед ним описатель и двоеточие. Тогда, хотя выполнение ядра и будет состоять из выполнения выражения, вид выполняемого выражения (а, следовательно, как мы увидим в гл. IV, и процесс его выполнения) будет диктоваться поставленным описателем.

Правда, превращать выражение в *void cast* не имеет особого смысла, так как роль такого превращения, как мы далее увидим, сводится просто к «выбрасыванию» полученного значения (если таковое было), а это и так при необходимости автоматически осуществляется. Но *void cast* в совокупности с аппаратом приведенных делает излишним понятие обозначения процедуры без параметров (в гл. IV увидим, как именно) и, кроме того, вместе с видовыми ядрами участвует в образовании обозначений процедур.

Следует отметить, что здесь проявляется один из основных принципов построения АЛГОЛе-68: как можно меньше вспомогательных, не имеющих самостоятельной роли понятий и как можно больше возможностей взаимодействия понятий. Этот принцип авторы языка называли *принципом ортогональности*, и мы еще будем иметь возможность указать на его проявление при рассмотрении других конструкций.

## § 8. Описатели

Конструкции, называемые в АЛГОЛе-68 *описателями* (*declarers*), являются обобщением таких составных частей описаний в АЛГОЛе-60, как *real*, *integer array*, *procedure* и т. п., т. е. частей, связанных с классом и типом величин. Правда, в АЛГОЛе-60 не было концепции вида, так что отсутствовали и средства спецификации (посредством описателей) всех на самом деле различаемых видов (например, возможных процедур). В АЛГОЛе-68 каждый вид может

быть специфицирован описателем. Более того, некоторые описатели специфицируют не только виды, но и другие свойства значений (например, в описателях кратных значений можно указать границы, различие же границ в паспорте кратного значения не является видовым). Описатели применяются в АЛГОЛе-68 более широко, чем в АЛГОЛе-60 и могут входить в состав не только описаний, но и выражений (например, они входят, как мы видели, в состав ядер). Более того, некоторые конструкции, являющиеся частным случаем унитарного выражения, могут состоять из одного описателя (эти конструкции называются *генераторами* и будут рассмотрены в следующем параграфе).

В настоящем параграфе мы рассмотрим строение описателей (специфицируемые ими виды будут в большинстве случаев очевидны, и мы не будем злоупотреблять педантизмом изложения), но сделаем это рассмотрение несколько упрощенным. Дело в том, что в состав описателей, кроме раз и навсегда зафиксированных символов, могут входить особые значки, не определяемые в языке и фиксируемые реализацией. Эти значки называются *индикантами вида*. С помощью специальной конструкции — *описания вида* — можно связать с таким значком определенный описатель, а стало быть, и вид. (В состав этого описателя снова могут входить индиканты вида.)

Употребление индикантов вида бывает удобно в целях сокращения записи, но также дает и принципиально новые возможности, которых мы позднее вкратце коснемся.

Сейчас мы рассмотрим только описатели, в состав которых не входят индиканты вида, причем ограничимся неформальным изложением.

Описатели делятся на три класса: *формальные, фактические и виртуальные*. Такое деление вызвано следующими соображениями. Как уже упоминалось, в описателях кратных видов имеется возможность указывать границы, т. е. задавать невидовую информацию. В некоторых случаях такая информация доступна и может быть соответствующим образом использована при выполнении (как мы далее увидим, это бывает, когда она лежит в известном смысле «на поверхности» описателя). Тогда ее разрешается писать. В других же случаях эта информация согласно процессу выполнения использована быть не может и все равно теряется. В таких случаях писать ее запрещается. Наконец, бывают случаи, когда указать границы

необходимо, ибо иначе процесс выполнения окажется по смыслу неопределенным. Тогда запрещается не писать границы.

Если в описателе вида кратного значения невидовую информацию можно писать, а можно и не писать, то мы имеем дело с формальным описателем, если границы писать обязательно — то с фактическим, если же их нельзя писать — с виртуальным. Вообще же синтаксис описателей устроен так, что путем комбинаций различных возможностей для формальных, фактических и виртуальных описателей разных видов и составления этих описателей друг из друга достигается то, что для каждого вида определено свое множество формальных, фактических и виртуальных описателей, удовлетворяющих вышеуказанным соображениям.

Смысл позиций формальных, фактических и виртуальных описателей мы выясним позже, а сейчас рассмотрим строение виртуальных описателей, которые специфицируют виды «чисто», без наличия невидовой информации в случае видов кратных значений (или видов, в состав которых входят виды кратных значений). Слово «виртуальный» будем при этом для краткости опускать (в состав виртуальных описателей никакие другие описатели, кроме виртуальных, входить не могут, так как внутри виртуального описателя невидовая информация тем более недоступна).

Описатели, специфицирующие простые (не составные) виды, очевидны — это следующие символы:

**int, real, bool, char, format.**

Описатели «длинных» видов образуются приписыванием нужного числа **long**, например:

**long int, long long real.**

Описатели видов процедурных значений начинаются с символа **proc**. За ним, если специфицируемый вид начинается с *procedure with* (т. е. является видом процедуры с параметрами), следуют заключенные в скобки описатели видов параметров (т. е. видов, входящих в состав вида процедурного значения). Эти описатели отделяются друг от друга запятой. Наконец, если в состав специфицируемого вида входит вид вырабатываемого значения (т. е. специфицируется вид функции), то после скобок помещается описатель этого вида.

Примеры:

```
proc (real, bool) int
```

```
proc (real)
```

```
proc int
```

```
proc
```

```
proc (proc (real) int, bool) proc (real, bool) int
```

```
proc (real) proc
```

```
proc proc proc
```

Заметим, что, несмотря на возможную сложную скобочную структуру, это — просто описатели, играющие такую же роль, как, скажем, *real*, только специфицирующие другие виды.

В АЛГОЛе-60 описание процедуры могло начинаться, например, следующим заголовком:

```
real procedure  $f(x, y)$ ; real  $x$ ; procedure  $y$ 
```

Эта конструкция — не описатель, здесь вводятся в рассмотрение три величины, информация о виде которых разорвана и неполна. В АЛГОЛе-68 аналогичное описание  $f$  могло бы начинаться, например, так (если  $y$  — идентификатор процедуры с одним целым параметром):

```
proc (real, proc (int)) real  $f = (real\ x, proc\ (int)\ y)$  real:
```

Справа от знака равенства здесь помещено начало обозначения процедуры.

В силу очевидной избыточности информации расширения позволяют записывать подобные тексты короче, в данном случае так:

```
proc  $f = (real\ x, proc\ (int)\ y)$  real:
```

Но если бы после знака равенства стояло не обозначение процедуры, а какое-нибудь другое выражение, например, идентификатор другой процедуры, то описатель нужно было бы выписать полностью. Насколько это может оказаться утомительным — предоставляем судить читателю.

Рассмотрим теперь описатели видов структурных значений. Они состоят из символа **struct** и заключенных в скобки *спецификаторов полей*. Каждый спецификатор состоит из описателя, специфицирующего вид соответствующего поля структурного значения, и названия этого поля — *выделителя*. Спецификаторы тоже отделяются друг от друга запятой.

Примеры:

**struct (real *re*, real *im*)**

(расширения „*„* позволяют в данном случае опустить второе вхождение **real**)

**struct (int *a*, proc (real) bool *b*)**

**struct (real *a*, struct (bool *a*, int *b*) *b*)**

Выделители должны быть различными на одном уровне, но на разных уровнях могут и совпадать (как в третьем примере), это не вызывает неоднозначностей при выполнении выделений полей. Например, если *s* — идентификатор вида, специфицируемого третьим описателем, то значением *b of s* будет поле значения *s*, само являющееся структурой, значением же *b of b of s* будет целое поле этой структуры.

Обратимся к описателям вида имен. Эти описатели состоят из символа **ref** и описателя вида того значения, на которое имя ссылается (т. е. вида, следующего за *reference to* в виде имени).

Примеры:

**ref real**

**ref proc (int)**

**ref struct (ref real *a*, ref bool *b*)**

**ref ref proc ref bool**

Наконец, описатели видов кратных значений начинаются со скобок [*и*], между которыми помещается на единицу меньше запятых, чем размерность массива, равная числу *row of* в начале специфицируемого вида; после скобок следует описатель вида, следующего за всеми этими *row of*.

Примеры:

**[,] real**

**[ ] ref struct (real *a*, ref [,] bool *b*)**

**[,] proc ([,] real) [ ] ref bool**

Позиции в скобках (разделяемые запятыми, если их больше одной) как раз и являются теми позициями, которые можно или нужно заполнять дополнительной информацией в случае неvirtуального описателя.

Мы сейчас рассмотрим только, что именно и когда пишется в таких позициях, и оставим выяснение смысла этих вставок до разбора семантики (т. е. выполнения) связанных с описателями конструкций.

В случае формального описателя писать можно следующее:

*formal lower bound, up to symbol, formal upper bound*  
где:

*formal LOWER bound: strict LOWER bound option, flexible symbol option; strict LOWER bound option, either symbol option.*

*strict LOWER bound: strong integral tertiary.*

*LOWER: lower; upper.*

(напомним, что *option* в конце понятия означает «может быть, а может и не быть»).

Примеры:

1 : m

1 : m flex

: m

1 :

: m either

: flex

either : flex

В случае фактического описателя обязательно нужно писать *жесткую* (*strict*) нижнюю и верхнюю границы, после которых можно помещать (а можно и не помещать) *flex*. Смысл границ содержательно ясен, символы *flex* и *either* связаны с *подвижной* границей (нулевыми значениями некоторых состояний).

Именно с помощью фактических описателей образуются выражения, называемые генераторами (вид генератора начинается с *reference to*, т. е. он обладает некоторым именем). Поэтому для фактического описателя определена семантика, т. е. процесс его выполнения (сколь это ни необычно). Из семантики становится ясным, какие описатели, входящие в состав фактического описателя, должны быть по синтаксису виртуальными.

Рассмотрим выполнение фактического описателя, опустив некоторые детали, к рассмотрению которых мы еще не подготовлены.

Фактический описатель выполняется следующим образом:

Шаг 1. Если описатель начинается с символа *struct*, то шаг 2, иначе, если он начинается с [, то шаг 3, в противном случае рассматривается некоторое значение того вида, который специфицирует данный описатель, и шаг 6.

Шаг 2. Все описатели, входящие в состав данного «структурного» описателя (а они по синтаксису — фактические), выполняются параллельно; те значения, на

которые ссылаются значения этих описателей (а они, как будет видно из дальнейшего, — имена), объявляются полями нового (и отныне рассматриваемого) структурного значения данного вида, и шаг 6.

Шаг 3. Все жесткие нижние и верхние границы, входящие в состав данного «кратного» описателя (те, которые заключены в индексные скобки, с которых он начинается), выполняются параллельно.

Шаг 4. Формируется паспорт (будущего кратного значения), состоящий из начала отсчета, равного 1, и столько-же пятюрок, сколько индексных позиций заключено в скобки в начале описателя; пусть их будет  $n$ , тогда для  $i = 1, \dots, n$ ,  $nl_i(u_i)$  полагается равным значению жесткой нижней (верхней) границы  $i$ -й из этих позиций; если после этой нижней (верхней) границы следует символ flex, то  $s_i(t_i)$  полагается равным 0, в противном случае  $s_i(t_i)$  полагается равным 1; далее шаг  $d_n$  полагается равным 1, и для  $i = n, n-1, \dots, 2$  шаг  $d_{i-1}$  полагается равным  $(u_i - l_i) \times d_i$ .

Шаг 5. Сформированный паспорт делается паспортом некоторого (и отныне рассматриваемого) кратного значения данного вида; элементы этого значения получаются следующим образом:

если описатель, следующий за индексными скобками, начинается с символа **struct** (а тогда он по синтаксису — фактический), то он выполняется столько раз, сколько должно быть элементов согласно паспорту, и  $i$ -й элемент — это значение, на которое ссылается  $i$ -е полученное имя (значение описателя);

в противном случае каждый элемент — это просто какое-то значение нужного вида.

Шаг 6. Создается имя, отличное от всех других имен, вид этого имени начинается с *reference to*, после которого следует вид, специфицируемый данным описателем; это созданное имя начинает ссылаться на рассматриваемое значение, это же имя становится значением данного описателя.

Прокомментируем приведенную семантику. Рассмотрим сначала случай, к которому в конечном счете все сводится, а именно, когда после первого шага сразу выполняется заключительный шестой (т. е. когда описатель не кратный и не структурный). Тогда выполнение заключается в создании нового имени, которое начинает ссылаться на «какое-то» значение специфицируемого вида. Нет средств

выяснить, какое это значение, оно нужно только для порядка: имя обязано на что-то ссылаться. В дальнейшем этому имени может быть присвоено нечто конкретное. (Аналогией может служить такой приказ: завести рабочую ячейку из числа свободных. Содержимое заведенной ячейки заранее не известно и поэтому вряд ли полезно, но в дальнейшем в нее можно будет заслать то, что нужно.)

Заметим, что если выполняемый описатель начинается с **gef** или с **rgos**, то в его состав может входить кратный описатель. Но мы видим, что процесс выполнения никак не реагирует на возможность наличия границ в этом описателе. Поэтому все описатели, стоящие в фактическом описателе после **gef** или **rgos**, по синтаксису виртуальные. (Можно, конечно, попытаться изменить семантику и попробовать извлекать границы из глубины описателя с тем, чтобы потом как-то их использовать. По-видимому, авторы строили синтаксис так, чтобы он предписывал задание границ в описателях лишь в тех случаях, когда выполнение описателя сопряжено при реализации с выделением участка памяти под соответствующий массив.)

Наоборот, в случае структурного фактического описателя все описатели, входящие в его состав (на внешнем уровне), — фактические, так как все они самостоятельно выполняются (параллельность выполнения должна уже стать привычной), и если какой-либо из них — кратный, то при его выполнении границы потребуются и будут использованы (так что проникновение вглубь описателя возможно только через структуры).

Рассмотрим теперь случай выполнения кратного фактического описателя.

Сначала выполняются все его жесткие нижние и верхние границы (они необходимы по смыслу, так как их значения используются на четвертом шаге, формально же их наличие обязательно, так как описатель фактический).

Из значений этих границ формируется паспорт следующим образом. Его начало отсчета полагается равным 1 (элементы вновь заводимых кратных значений всегда нумеруются от единицы, другие начала отсчета могут появляться, как мы видели, при выполнении вырезов).

Границы в пятерках паспорта полагаются равными значениям данных границ, а шаги вычисляются по этим значениям (при этом  $d_n$  всегда полагается равным 1).

Наконец, значения состояний (0 или 1) зависят от наличия или отсутствия символа *flex* в соответствующей позиции описателя.

Далее этот паспорт становится паспортом кратного значения, элементы которого в общем случае — это просто «какие-то» значения нужного вида. Эти элементы потом могут быть замещены путем присваивания, но здесь существенно, что паспорт фиксирован, и не всякое присваивание окажется законным: при выполнении присваивания, как мы видели, происходит сверка границ и состояний.

Если после индексных скобок стоит неструктурный описатель, то в его состав не должны входить кратные описатели с границами, т. е. погоня за границами прекращается. В случае же структурного описателя, стоящего после скобок, появляется легкая возможность гнаться за границами дальше путем самостоятельного выполнения его составных частей. Поэтому по синтаксису такой описатель — фактический, и пятый шаг делает соответствующий анализ. В любом случае дело завершается шестым шагом, на котором создается новое имя, начинающее ссылаться на полученное любым из рассмотренных способов значение.

Таким образом, мы выяснили роль жестких верхних и нижних границ и символа *flex* в индексных позициях кратных фактических описателей. Что касается роли информации, которую можно помещать в таких позициях в формальных описателях, то ее мы рассмотрим в § 10 при разборе конструкции, называемой *описанием идентичности*.

## § 9. Генераторы

Конструкции, называемые *генераторами* (*generators*), относятся к классу вторичных выражений (как уже упоминалось, в дальнейшем будет приведена синтаксическая иерархия унитарных выражений).

Генератор состоит из фактического описателя, перед которым может стоять, а может и не стоять символ *loc* (*local symbol*). В первом случае генератор называется *локальным*, во втором — *глобальным*.

Выполнение любого генератора заключается в выполнении его фактического описателя, и значение генератора — значение этого описателя. (Так как значения фак-

тических описателей — имена, то вид генератора должен начинаться с *reference to*, поэтому всюду, где в правилах синтаксиса фигурирует *MODE generator* и тем более *MOID generator*, возможны тупики.)

Таким образом, генератор называется генератором потому, что при его выполнении создается («генерируется») новое имя и значение (неопределенное), на которое это имя ссылается.

Это новое имя нужно снабдить областью действия (см. § 7 гл. I). Область действия определяется по-разному для локальных и глобальных генераторов.

Областью действия имени, созданного локальным генератором, объявляется наименьший *уровень* (*range*), содержащий этот генератор. Уровень — это либо обозначение процедуры, либо синтаксическая единица еще одного класса — *последовательное выражение* (*serial MOID clause*). Последовательное выражение — это то, что заключено в скобки в закрытом выражении, но из чего оно состоит, мы еще не знаем (в частности, это может быть любое унитарное выражение и, следовательно, любой из рассмотренных нами его частных случаев).

В случае глобального генератора областью действия имени считается вся программа.

Ясно, что при реализации эта разница в областях действия сведется к разному использованию памяти. Действительно, после завершения выполнения какого-либо уровня можно освободить всю память, связанную с его локальными именами (т. е. именами, для которых этот уровень является областью действия), и использовать эту память при выполнении других уровней. Память же, связанная с глобальными именами, как правило, не может быть освобождена в течение выполнения всей программы \*). Поэтому глобальных генераторов следует по возможности избегать.

Заметим, что так как генератор относится к классу вторичных выражений, то он может быть как левой, так и правой частью присваивания.

---

\*) Однако, если некоторое глобальное имя не присвоено никакому другому имени и им не обладает никакой идентификатор, то память, связанную с этим глобальным именем, освободить все же можно. Процесс исследования глобальных имен на предмет установления возможности освобождения связанной с ним памяти называется «уборкой мусора». Эта неприятная работа предпринимается обычно тогда, когда ощущается острая нехватка памяти.

Например:

1) `loc real := 3.14`

2) `xx := real`

В первом примере вновь созданному имени присваивается значение 3.14. Подобные конструкции широко используются, так как есть возможность (с помощью описания идентичности, см. § 10) связать это имя с идентификатором.

Во втором примере имя, которым обладает `xx` (оно должно быть именем имени, т. е. его вид должен быть *reference to reference to real*, так как само присваиваемое значение — имя), начнет ссылаться на вновь созданное имя (ссылающееся в свою очередь на неопределенное значение вида *real*). Продолжая пример, можно написать следующее присваивание:

`(ref real : xx) := 1`

(Ядро заключено в скобки, так как само оно не относится к третичным выражениям. Отсутствие скобок было бы эквивалентно записи

`ref real : (xx := 1)`

а включение ядер в третичные выражения сделало бы запись без скобок двусмысленной.)

Тогда значением ядра будет имя, созданное перед этим с помощью генератора, и этому имени будет присвоено значение 1.

Таким образом, здесь вновь созданное имя не связано с самостоятельным идентификатором и служит только промежуточным звеном в цепочке ссылок (косвенных адресов).

## § 10. Описания идентичности

Отвлечемся на некоторое время от выражений и, используя накопленный материал, рассмотрим основной тип описаний в АЛГОЛе-68, намного перекрывающий возможности всех описаний АЛГОЛа-60 — *описание идентичности (identity declaration)*.

(Роль описаний в АЛГОЛе-68 аналогична роли описаний в АЛГОЛе-60: посредством описаний вводятся в рассмотрение и наделяются определенными свойствами идентификаторы и некоторые другие внешние объекты.

Но в АЛГОЛе-68 описания используются более широко и свободно, они могут входить в состав выражений, в определенных пределах чередоваться с операторами и т. п. При рассмотрении строения крупных синтаксических единиц в § 13 мы это увидим.)

Несмотря на такую мощь, синтаксис описания идентичности очень прост. Это объясняется тем, что у нас уже имеются мощные заготовки, и для получения желаемого эффекта (т. е. возможности формулирования подходящей семантики) их достаточно простым образом скомбинировать. Правда, сама семантика не получается особо простой, так как опять много хлопот причиняют кратные значения, так что мы и на этот раз будем вынуждены ее прокомментировать:

*identity declaration: formal MODE parameter,  
equals symbol,  
actual MODE parameter.*

*formal MODE parameter: formal MODE declarer,  
MODE mode identifier.*

*actual MODE parameter: strong unitary MODE clause.*

(Из последнего правила мы изъяли еще одно прямое порождение, связанное с форматами, так как форматы мы не рассматриваем.)

Второе правило уже встречалось при рассмотрении обозначений процедур. Это и естественно, так как при преобразовании обозначения процедуры в рутину каждый формальный параметр дает начало описанию идентичности, в котором он стоит слева от знака равенства (*equals symbol*). Таким образом, формальные и фактические параметры в АЛГОЛе-68 — это не монополия аппарата процедур, как в АЛГОЛе-60, они используются самостоятельно в общих языковых конструкциях (ортогональность!).

Примеры:

```
int i = 1
ref real x = loc real
```

Описатели, помещаемые в левой части описания идентичности, специфицируют вид описываемого идентификатора и, кроме того, могут предписывать его значению некоторые невидовые свойства — границы и состояния (т. е. подвижность границ). В АЛГОЛе-60 в описаниях массивов тоже задавались границы, причем в обязательном порядке и без возможности их смены (не считая, конечно,

повторных входов в блок). Здесь такой жесткости нет — можно не задавать ни границ, ни состояний. Тогда они будут такими, какими получатся в результате выполнения фактического параметра. Поэтому описатели, стоящие в левой части описания идентичности, синтаксически относятся к классу формальных. Не задавать границ и состояний в формальных описателях можно всегда, задавать же их разрешается только тогда, когда эта информация согласно семантике может быть использована. Синтаксически это достигается тем, что некоторые описатели, входящие в состав формальных, являются виртуальными. В каких случаях это бывает, а также смысл соответствующих запретов мы выясним при разборе нижеследующей семантики описания идентичности.

Описание идентичности выполняется следующим образом (с точностью до некоторых устраненных для простоты деталей).

Шаг 1. Его фактический параметр и все жесткие нижние и верхние границы, содержащиеся в его формальном параметре (на всех уровнях строения формального описателя), выполняются параллельно.

Шаг 2. Пусть  $v$  — значение фактического параметра, тогда полагается:

$$w = \begin{cases} \text{значению, на которое ссылается } v, & \text{если } v \text{ — имя,} \\ \text{самому } v \text{ — в противном случае.} \end{cases}$$

Шаг 3. Если  $w$  — не кратное и не структурное значение, то шаг 6, иначе, если  $v$  — не имя, то шаг 5.

Шаг 4. Для каждой формальной нижней и верхней границы, содержащейся в формальном параметре: если граница содержит символ `flex` (не содержит ни `flex`, ни `either`) и соответствующее состояние в паспорте  $w$  равно 1 (0), то дальнейшее выполнение не определено.

Шаг 5. Для каждой жесткой нижней и верхней границы, содержащейся в формальном параметре: если после границы не следует `flex` и ее значение не совпадает с соответствующей границей в паспорте  $w$ , то дальнейшее выполнение не определено.

Шаг 6. Идентификатор формального параметра начинает обладать значением  $v$  фактического параметра, и выполнение завершено.

Прокомментируем приведенную программу выполнения. Первый шаг вряд ли нуждается в комментариях. Существенно только, что выполняются все жесткие гра-

ницы, содержащиеся в формальном описателе по ходу его построения с помощью других описателей, а не только «внешние». Это естественно: все, что разрешается писать в формальном описателе, должно быть по возможности использовано. (Правда, в состав выражений, задающих жесткие границы, вообще говоря, снова могут входить описания идентичности, содержащие границы; такие границы, конечно, не учитываются и будут выполняться лишь тогда, когда этого потребует процесс выполнения «настоящих» границ.)

Если формальный описатель не связан с видами кратных значений, т. е. никаких границ в нем нет, то на первом шаге просто выполняется фактический параметр, и дальнейшее выполнение сводится к тому, что идентификатор формального параметра начинает обладать полученным значением.

Таким образом, в общем случае описание идентичности выполняется совсем просто, и вся разветвленная логика описывает только работу с описателями, содержащими границы. Рассмотрим, что и в каких случаях здесь предусмотрено. Возможны два случая.

Первый случай — вид, специфицируемый формальным описателем, — вид имени (т. е. описатель начинается с *ref*). В этом случае фактическим параметром может быть генератор, выполнение которого, как мы знаем, заключается в выполнении его фактического описателя. Предположим, например, что конструкция *ref ref [1 : n flex] real* является формальным описателем в таком описании идентичности. Тогда соответствующим генератором не может быть, скажем, *ref [1 : n] real*, так как такой описатель не является по синтаксису фактическим (почему это так, мы разбирали при рассмотрении фактических описателей). Подходящим фактическим описателем является *ref [ ] real*. Но его значением является имя, ссылающееся на имя, в свою очередь ссылающееся на некоторое, неизвестно какое значение вида *row of real* (см. § 8, выполнение фактического описателя). Сверка границ и состояний такого значения с чем-то конкретным, диктуемым формальным описателем, была бы бессмысленной. Поэтому синтаксис формальных описателей устроен так, что информацию о границах и состояниях в них в случае видов имен можно писать только в тех индексных позициях, которые соответствуют индексным позициям возможных фактических описателей. Например, описатель,

следующий в формальном описателе после двух *ref*, является по синтаксису виртуальным, так что вышеприведенная конструкция *ref ref [1 : n flex] real* не является формальным описателем (и вообще описателем, так как множество формальных описателей шире множества фактических и виртуальных).

Таким синтаксическим согласованием формальных и фактических описателей завершается борьба с трудностями, связанными с невидовой информацией в описателях, борьба, потребовавшая деления описателей на несколько классов и значительного усложнения их синтаксиса и выполнения связанных с ними конструкций.

Рассмотрим теперь само выполнение описания идентичности в случае, когда формальный описатель специфицирует вид имени.

Если имя этого вида ссылается на простое, процедурное значение или имя, то в формальном описателе не может быть формальных границ: в случае простого значения границы вообще отсутствуют, в случае процедурного значения или имени в силу согласования синтаксиса формальных и фактических параметров границы могут быть только виртуальными (т. е. в виртуальных описателях). Поэтому в этих случаях четвертый и пятый шаги работали бы вхолостую (но их выполнение потребовало бы времени для просмотра описателя). Поэтому в третьем шаге предусмотрен оптимизирующий анализ, отсылающий в вышеуказанных случаях сразу к заключительному шестому шагу.

Если же имя данного вида ссылается на кратное или структурное значение, то формальные границы есть (в случае кратного значения) или могут быть (в случае структурного значения), и четвертый и пятый шаги производят сверку состояний и границ, заключающуюся в следующем.

Значение фактического параметра по синтаксису всегда того вида, который специфицирует формальный описатель. Поэтому каждой индексной позиции формального описателя соответствует в паспорте этого значения некоторая граница и некоторое состояние.

Наличие в формальной границе символа *flex* (*flexible*—подвижный, гибкий) означает, что мы хотим, чтобы в дальнейшем при присваивании нашему имени другого значения соответствующую границу можно было бы менять. Следовательно, соответствующее состояние должно быть равно 0. Поэтому проверяется, действительно ли это так. Если

это не так, то можно поступить двояко: или положить-таки состояние равным 0, или же считать такую ситуацию ошибочной. Авторы АЛГОЛа-68 идут по второму пути, резонно считая, что такого рода информация должна быть в формальном и фактическом параметре согласована. Например, если фактический параметр — генератор, то подобная ситуация означает, что в формальном описателе некоторая индексная позиция снабжена подвижной границей, а соответствующая позиция в фактическом описателе — нет, и спрашивается, чему же верить? Поэтому дальнейшее выполнение в таких случаях объявляется неопределенным.

Наличие в формальной границе символа *either* (любой) означает, что нам безразлично, чему равно соответствующее состояние: если оно равно 0, значит, можно будет менять соответствующую границу, а нет, так и не надо (например, если мы заведомо знаем, что смен границы не будет, а каково состояние — не знаем, то следует на всякий случай поставить *either*, так как иначе, если состояние равно 0, то дальнейшее выполнение будет не определено, см. дальше).

Наконец, отсутствие в формальной границе и *flex* и *either* означает, что мы запрещаем смену этой границы. Поэтому аналогично первому случаю производится проверка, равно ли соответствующее состояние 1, и если нет, то дальнейшее выполнение объявляется неопределенным.

На пятом шаге осуществляется сверка границ.

Если после жесткой границы в формальном описателе не стоит *flex*, то это означает, что сейчас требуется именно такая соответствующая граница в значении (хотя, если на месте *flex* стоит *either* и соответствующее состояние равно 0, то она может в дальнейшем измениться). Если это не так, то дальнейшее выполнение не определено. (Заметим, что при такой сверке границ теряет смысл помещение жесткой границы перед *flex*, так как она, в отличие от таких конструкций в фактических описателях, все равно не будет использована. Однако синтаксически это допустимо. Казалось бы, запретить ставить в формальных описателях жесткие границы перед *flex* легко: достаточно, например, так изменить синтаксис формальной границы:

*formal LOWER bound: strict LOWER bound option,  
either symbol option;  
flexible symbol.*

Но на самом деле это не приведет к цели из-за индикантов вида, которые могут входить в состав описателей и будут рассмотрены нами в § 12.)

Если все в порядке, то на шестом шаге устанавливается отношение «обладать» между идентификатором формального параметра (т. е. описываемым идентификатором) и значением фактического параметра, и выполнение завершается.

Вхождение идентификатора в формальном параметре является определяющим, так что отныне будет определено выполнение соответствующего примененного вхождения этого идентификатора как базы. О том, что такое вхождение идентификатора — определяющее, мы уже упоминали при рассмотрении обозначений процедур, но теперь должно быть ясно, почему это так.

Второй случай — вид, специфицируемый формальным описателем, — не вид имени.

В этом случае первый анализ третьего шага имеет тот же оптимизирующий смысл и отсылает к шестому шагу, если значение фактического параметра простое или процедурное. (По синтаксису описатель, следующий в формальном описателе после **procs**, — виртуальный. Так сделано, по-видимому, в целях простоты и единообразия, но, вообще говоря, можно представить себе осмысленное использование информации о границах в таких описателях.)

Тот же третий шаг делает обход сверки состояний (т. е. четвертого шага), так как не имени ничего нельзя присвоить и, следовательно, значения состояний несутельственны. Поэтому символы **flex** и **either** лишены в этой части своего смысла.

Символ **flex** еще нужен в пятом шаге, а помещение **either** в этом случае лишено всякого смысла, хотя формально ставить его можно. Здесь, таким образом, тоже запрещено не все, что не имеет смысла. Но, по-видимому, и нецелесообразно добиваться такого запрещения, так как оно неизбежно приводит к еще большему усложнению синтаксиса описателей.

Пятый и шестой шаги выполняются, как было рассмотрено выше, и выполнение опять-таки завершается (если завершается!) тем, что идентификатор начинает обладать значением фактического параметра.

Рассмотрим теперь на примерах, как перекрываются описанием идентичности возможности описаний АЛГОЛа-60.

## АЛГОЛ-60:

- 1) описание типа

**real x**

- 2) описание массива

**array a [1:10, 1:10]**

- 3) описание процедуры

**real procedure f (x); real x; f := if x > 0 then x else - x**

- 4) описание переключателя

**switch s := l1, l2, l3**

## АЛГОЛ-68:

- а) Без использования расширений:

1) **ref real x = loc real**

2) **ref [,] real a = loc [1:10, 1:10] real**

3) **proc (real) real f = ((real x) real: if x > 0 then x else - x fi)**

4) **[ ] proc s = (go to l1, go to l2, go to l3)**

(и  $s[i]$  вместо **go to  $s[i]$**  в АЛГОЛе-60).

- б) В расширенном языке:

1) **real x**

2) **[1:10, 1:10] real a**

3) **proc f = (real x) real: if x > 0 then x else - x fi**

4) то же, что и в а).

Как мы видим, в расширенном языке привычные вещи могут быть записаны достаточно коротко (заметим, что при использовании глобальных генераторов такие преимущества сокращенной записи, как правило, не предоставляются, чем подчеркивается нежелательность таких генераторов).

Первые два описания можно усилить, сделав фактическим параметром не просто генератор, а присваивание с генератором в левой части, например:

**ref real x = loc real := 3.14**

(в расширенном языке

**real x := 3.14)**

После выполнения этого описания идентификатор  $x$  будет обладать именем, ссылающимся на значение 3.14

(по старой терминологии «переменная  $x$  получит начальное значение»). Заметим, что в этом примере ярко проявляется принцип ортогональности. Действительно, по ходу выполнения здесь создается имя, ему что-то присваивается, и этим именем начинает обладать идентификатор.

В [1] для достижения такого эффекта еще привлекалось довольно много вспомогательных понятий, загромождавших синтаксис и семантику как описаний идентичности, так и генераторов. Здесь же не потребовалось ничего вспомогательного: фактическим параметром является частный случай унитарного выражения — присваивание, в левой части которого тоже стоит частный случай унитарного выражения — генератор. Изящно, не правда ли? Может показаться утомительной некоторая длиннота такой записи, но, конечно, следует пользоваться расширениями, не забывая, однако, какая полная запись при этом подразумевается, и как она трактуется.

В третьем описании в расширенном языке можно было бы вместо знака равенства поставить знак присваивания (предлагаем посмотреть, каким изменениям полной записи это соответствует), тогда  $f$  будет обладать именем, ссылающимся на процедурное значение; в дальнейшем этому имени можно будет присвоить другое значение, в то время как в нашем примере идентификатор  $f$  неразрывно связан с одной и той же рутинной.

Точно так же, если написать

$$\text{real } x = 3.14159$$

то  $x$  будет уже не переменной, а константой, т. е. будет неизменно обладать одним и тем же числовым значением; если же желательно, чтобы  $x$  был переменной (обладал именем), то, как мы только что видели, достаточно вместо  $=$  поставить  $:=$ .

Четвертый пример формально еще не должен быть понятен (см. § 2 гл. IV), и мы рассчитываем на его неформальное восприятие. Заметим, что и здесь, переходя к именам и получая возможность присваивания новых значений (рутин) элементам массива процедур, мы получим еще более сильный аппарат переходов. Но ясно, что даже и без этого здесь могут быть охвачены все возможности, которые дают переключатели в АЛГОЛе-60 (поэтому самостоятельное понятие, аналогичное переключателю, не вошло в АЛГОЛ-68).

## § 11. Формулы и описания операций и приоритетов

До сих пор нам не встречались столь привычные по АЛГОЛу-60 формулы, т. е. конструкции со знаками операций (такие, например, как  $a + b$  или  $\neg x \vee y$ ). Но, конечно, в АЛГОЛе-68 тоже есть формулы, и они внешне сходны со старыми. Однако это сходство обманчиво — на самом деле в образовании и использовании формул между АЛГОЛом-68 и АЛГОЛом-60 имеются большие различия в сторону обобщения и уточнения.

В АЛГОЛе-68 символы, могущие быть знаками операций (это — некоторые конкретные символы, такие, например, как  $+$  или  $-$ , а также не определяемое в языке и фиксируемое реализацией множество *индикантов операций*), как и идентификаторы, не имеют неизменного смысла и тоже вводятся в рассмотрение посредством специальных описаний — *описаний операций* (*operation declarations*). Это не значит, конечно, что каждый раз при написании программы нужно заботиться об обычном смысле, скажем, знаков  $+$  или  $-$  (да и как выразить смысл  $-$ , он ведь первичен и не выражается языковыми средствами). Дело в том, что имеется большой набор стандартных описаний, в частности, описаний операций (причем такие первичные понятия, как, например, вычитание, описываются в конечном счете на естественном языке). Считается, что программа, написанная программистом, — это часть общей программы, в которую входят стандартные описания, так что все, что в них содержится, может не описываться и будет тогда пониматься в стандартном смысле. Набор стандартных операций в АЛГОЛе-68 (для них и заведены конкретные символы) включает привычные арифметические и логические операции и операции отношения, но имеет еще много сверх того.

Ничто не мешает переописать какую-либо операцию, в частности, стандартную, и тогда она, как и стандартный идентификатор (есть и такие, например,  $\sin$ ), временно приобретет другой смысл.

Более того, в отличие от идентификаторов один и тот же знак операции можно описывать на одном уровне более одного раза (но только по-разному в смысле количества и видов операндов, чтобы можно было различить, что именно используется). Наконец (и это логическое следствие предыдущего), если уже описанный знак операции описывается на внутреннем уровне еще раз и совсем

по-другому, т. е. для операндов других видов, то старый смысл не уничтожается. Например, если мы описали умножение матриц и употребили для этого тот же знак  $\times$ , что и для умножения вещественных чисел, то в записи  $a \times b$  знак  $\times$  будет пониматься по-разному в зависимости от видов  $a$  и  $b$ . (Переописав же  $\times$  для вещественных операндов, мы потеряем «обычное» умножение.) В стандартных описаниях операций такая ситуация типична, и некоторые знаки (например,  $=$ ,  $<$ ,  $\uparrow$  и др.) описываются по многу раз в разных смыслах.

Наличие такой вольности связано с укоренившейся у математиков привычкой называть действия над новыми объектами, обобщающие или в каком-то смысле аналогичные действиям над старыми объектами, старыми именами (умножение матриц, сложение кодов и т. п.) и обозначать их по-старому. Это действительно привычно и потому удобно, однако платить за такую возможность приходится довольно дорого, так как существенно усложняется синтаксический анализ. Но это уже забота не пользователей, а тех, кто реализует язык.

Что же представляют собой описания операций?

Различаются *бинарные* (*dyadic*) и *унарные* (*monadic*) операции. Бинарная операция означает выполнение каких-то действий над двумя операндами, унарная — над одним. Каким образом можно изобразить эти действия? Напрашивается ответ: с помощью соответствующим образом сформированных рутин, могущих затем быть вызванными, как это имело место в общем случае работы с рутинными. Таким образом, операция рассматривается как частный случай процедуры с одним или двумя параметрами, которая только вызываться будет не обычными вызовами, а с помощью других конструкций — формул. Поэтому описание операции построено аналогично описанию идентичности, только в нем возможны не все виды фактического параметра (т. е. части описания, стоящей справа от знака равенства), а лишь те, которые могут соответствовать операциям.

Описание операции получается из соответствующего описания идентичности, если заменить в нем идентификатор формального параметра на знак операции, убрать из формального описателя начальный символ `proc` (так как описатель обязан начинаться с `proc`, то незачем его особо писать) и приписать слева (т. е. вместо `proc`) символ *op* (*operation symbol*).

Пример. Описание идентичности:

1) без использования расширений:

```
proc (bool, bool) bool and = ((bool a, bool b) bool :  
    if a then b else false fi)
```

2) в расширенном языке:

```
proc and = (bool a, b) bool : if a then b else false fi
```

Описание операции:

1) без использования расширений:

```
op (bool, bool) bool  $\wedge$  =  
((bool a, bool b) bool : if a then b else false fi)
```

2) в расширенном языке:

```
op  $\wedge$  = (bool a, b) bool : if a then b else false fi
```

Выполняется описание операции так же, как выполнялось бы соответствующее описание идентичности, т. е. выполняется его фактический параметр, значение которого — рутина, и знак операции начинает обладать этой рутинной.

Такое вхождение знака операции в описание операции тоже является определяющим, и примененные вхождения должны идентифицировать его (правила идентификации здесь несколько сложнее, чем для идентификаторов, и предусматривают проверку соответствия видов операндов из-за того, что один знак может быть описан несколько раз).

Рассмотрим теперь сами формулы:

Формулы относятся к классу третичных выражений и могут быть любого вида или без вида (порождаются *MOID ADIC formula*, что такое *ADIC*, будет сказано дальше). Различаются *унарные* и *бинарные* формулы.

Унарная формула вида *M* состоит из знака унарной операции вида *M* (т. е. в описании этой операции после **op** следует один описатель в скобках, специфицирующий некоторый вид *M*<sub>1</sub>, а после скобок стоит описатель, специфицирующий вид *M*) и следующего за ним операнда того вида, который требуется согласно описанию операции, т. е. вида *M*<sub>1</sub> (как и в случае вызовов, не следует путать виды операндов, т. е. того, над чем производятся действия, и вида самой формулы, т. е. вида ее значения).

В случае формулы без вида (т. е. в случае *void*) в левой части описания операции отсутствует описатель после скобок (аналогично случаю *void call*).

Операндом унарной формулы может быть либо вторичное выражение, либо снова унарная формула (поэтому, в отличие от АЛГОЛа-60, можно писать несколько унарных операций подряд, например  $\neg \neg \neg a$ ).

Бинарная формула состоит из двух операндов, разделенных знаком бинарной операции (с видами здесь аналогичная картина). Операндом бинарной формулы может быть либо вторичное выражение, либо унарная формула, либо снова бинарная формула, т. е. приходим к обычному строению формул, например:

$$a + b \times c < d \vee \neg \text{bool}.$$

(Но заметим, что, в отличие от АЛГОЛа-60, например, формула  $-a \uparrow 2$  трактуется как  $(-a) \uparrow 2$ , так как  $-a$  является унарной формулой.)

Выполняется формула так же, как выполнялся бы соответствующий вызов: рассматривается копия рутины, которой обладает определяющее вхождение знака операции, все *skip* (их один или два) заменяются соответствующими операндами (возможно, снова формулами, подобно тому как фактический параметр в вызове в свою очередь может быть вызовом) и далее по семантике вызовов.

Мы видим, что нет принципиальной необходимости описывать операции: можно обойтись вызовами и описывать только идентификаторы. Однако ясно, что, например, вышеприведенная запись формулы намного более удобна и наглядна, чем такая:

$$\text{or (less (plus (a, mult (b, c)), d), not (bool))}$$

так что использование формул дает просто удобство привычной формы записи.

После того, как мы привели «процедурную» форму записи, эквивалентную ранее приведенной «формульной», внимательный читатель должен был ощутить некоторое недоумение. Действительно, в процедурной записи явно выписаны параметры вызовов. В формульной записи это означает, что для каждой операции именно так определены операнды. Но почему именно так, а не иначе, скобки ведь отсутствуют? Почему формулу  $a + b \times c$  мы трактуем

как  $a + (b \times c)$ , а не как  $(a + b) \times c$ ? Сейчас мы рассмотрим этот вопрос.

В АЛГОЛе-60 набор знаков операций был зафиксирован, поэтому там достаточно было раз и навсегда задать порядок действий: сначала  $\uparrow$ , затем  $\times$  и  $/$  и т. д.

Поэтому трактовка  $a + b \times c$  становилась однозначной.

Здесь мы так поступить не можем, потому что набор знаков операций и их смысл не фиксируются. Следовательно, необходимо специальное средство, с помощью которого можно было бы обеспечить однозначность. В АЛГОЛе-68 это средство заключается в следующем. С каждым знаком операции связывается некоторое положительное целое число, называемое *приоритетом* данной операции. Для бинарных операций это производится с помощью специального описания — *описания приоритета* (*priority declaration*), которое мы ниже рассмотрим. Унарным операциям всегда приписывается один и тот же приоритет — наивысший из всех возможных, и описания приоритетов для них отсутствуют.

В правилах синтаксиса фигурируют не просто формулы, но «унарные формулы» и «бинарные формулы такого-то приоритета». В *MOID ADIC formula* метапонятие *ADIC* как раз несет эту нагрузку.

Синтаксис устроен так, что по приоритетам операций обеспечивается однозначная трактовка формул, содержащих несколько знаков операций.

В стандартных описаниях для всех стандартных знаков бинарных операций заданы их стандартные приоритеты, которые и приводят к вышерассмотренной обычной трактовке.

Описание приоритета состоит из символа **priority**, знака операции, знака равенства и цифры от 1 до 9 включительно (унарным операциям всегда приписывается приоритет, равный десяти).

Например:

**priority** + = 6

**priority** = = 4

Вхождение знака операции в описание приоритета тоже является определяющим (оно несет часть информации об объекте), и все другие вхождения этого знака (в том числе и вхождение в описание операции) должны

идентифицировать его. Этим роль описания приоритета исчерпывается, и его выполнение не влечет никаких действий. Это и понятно, так как информация, которую несет описание приоритета (в отличие от описаний идентичности и операций), используется только при синтаксическом анализе программы, а не при ее выполнении.

Переопределять приоритет одной и той же операции запрещается (т. е. запрещается ситуация, когда в разных частях программы одна и та же операция имеет разные приоритеты). Но, конечно, возможен случай, когда несколько операций с одним и тем же знаком имеют разные приоритеты — для этого достаточно поместить разные описания операций и приоритетов на разных уровнях, т. е. таким образом, чтобы правила идентификации делали пустой область совместного использования описанных операций.

Одно описание приоритета может относиться к нескольким описаниям операций с одним и тем же знаком.

## § 12. Индиканты вида и описания вида

При рассмотрении описателей мы не рассматривали индикантов вида, которые могут входить в их состав. Сейчас мы займемся ими.

Как и в случае знаков операций, имеется некоторое количество стандартных индикантов вида (например, `string`, `compl` и `bits`), и, кроме того, считается, что реализацией фиксируется еще какое-то их множество. Мы, употребляя что-либо в роли индиканта вида, будем считать, что такой индикант допустим.

Индикант вида вводится в рассмотрение посредством еще одного описания — *описания вида* (*mode declaration*).

Стандартные индиканты содержатся в стандартных описаниях вида; как и в случае операций, ничто не мешает их переописывать.

Описание вида связывает с индикантом описатель (и тем самым вид, который этот описатель специфицирует). Состоит оно из символа `mode` (*mode symbol*) описываемого индиканта, знака равенства и фактического описателя. Например, `string` и `compl` описываются в стандартных описаниях следующим образом:

```
mode string = [1:0 flex] char
```

```
mode compl = struct (real re, real im)
```

(В расширенном языке второе описание может быть записано короче:

**struct compl = (real *re*, *im*)**

что и сделано в стандартных описаниях.)

Индикант **bits** описан в [1] так:

**mode bits = [1 : *bits width*] bool**

где *bits width* — стандартный идентификатор, значение которого — диапазон битовых значений единичной длины (он запрашивается у реализации) \*).

Выполнение описания вида не влечет никаких действий, оно играет роль только при работе с описателями во время анализа текста программы.

Индикант вида по синтаксису — частный случай описателя (формального, фактического и виртуального). Поэтому произвольный описатель может или не содержать индикантов, или быть индикантом, или, наконец, не будучи сам индикантом, иметь их в своем составе.

Например:

**[1 : 0 flex] char  
string  
struct (string *s*, bits *b*)**

---

\*) Как уже говорилось в § 2 при рассмотрении битовых обозначений, в [2] принята другая трактовка битов.

Это изменение трактовки объясняется очень простыми причинами. Если рассматривать битовое значение как массив логических значений, то выборка отдельного элемента такого значения должна реализоваться совсем иными средствами, чем в случае обычного массива, где каждый элемент занимает одну (или несколько) ячеек. Еще сложнее образовывать из такого массива вырезки, состоящие из нескольких элементов. При новой трактовке применить к битовым значениям общий аппарат вырезок невозможно, а для выборки отдельных элементов битового значения введена специальная операция.

Представляет интерес механизм, использованный в новой трактовке. Вид «битовый» — это структурный вид с одним полем вида «одномерный логический массив». Для выделения этого единственного поля используется идентификатор, состоящий из одной буквы, не имеющей внешнего представления. Поэтому само поле программисту недоступно. Над битовыми значениями в программе можно выполнять только те операции, которые определены в стандартных описаниях (а также те, которые можно определить через эти операции).

Аналогичный подход принят и к *слововым* значениям, представляющим собой строки, упакованные по несколько литер в ячейке.

Над описателями в связи с индикантами определяется специальная операция *развертки*, которая упрощенно заключается в следующем.

Если описатель содержит индикант, который по синтаксису этого описателя представляет собой фактический или формальный описатель, то этот индикант заменяется фактическим описателем из его описания вида, и развертка начинается *снова*; если таких индикантов нет, то развертка завершена.

Выполнение фактического описателя всегда начинается с того, что он развертывается. Точно так же и выполнение описания идентичности начинается с развертки его формального описателя.

Развертка производится для извлечения из описаний вида той информации, которая при дальнейшем выполнении может быть использована. Пусть, например, имеются описания вида:

$$\begin{aligned}\text{mode array} &= [1:m] \text{ real} \\ \text{mode row} &= [1:n \text{ flex}] \text{ real}\end{aligned}$$

и следующее описание идентичности:

$$\text{ref row } v = \text{loc array}$$

(описатели *array* и *row* специфицируют один и тот же вид — *row of real*, так что такое описание законно).

Если не производить развертки, то значением *v* после выполнения этого описания будет имя, ссылающееся на некоторое (неизвестно какое) значение вида *row of real*, т.е. информация о границах, содержащаяся в описаниях вида, будет утеряна. После развертки же формального описателя описание как бы станет таким:

$$\text{ref } [1:n \text{ flex}] \text{ real } v = \text{loc array}$$

(отсюда, кстати, видно, что запрет *n flex* путем одного только изменения синтаксиса формальных границ не проходит, так как есть возможность «протаскивания» таких конструкций из фактических описателей через индиканты).

Выполнение фактического описателя *array* начнется с его развертки, после чего выполнение пойдет так, как если бы было написано такое описание:

$$\text{ref } [1:n \text{ flex}] \text{ real } v = \text{loc } [1:m] \text{ real}$$

т.е. со всеми сверками границ и состояний, и результат этого выполнения окажется неопределенным, так как информация о подвижности верхней границы не согласована. Индиканты, представляющие собой виртуальные описатели, при развертке не заменяются, так как даже если в их описаниях вида и есть информация о границах, то здесь она все равно использована не будет (и даже замена индиканта таким описателем синтаксически недопустима). Например, пусть есть следующее описание идентичности:

ref ref row *rv* = loc ref row

(в расширенном языке просто *ref row rv*).

Это описание правильное, так как по синтаксису в формальном описателе после двух *ref* следует виртуальный описатель, значит, при развертке индикант *row* не заменится на *[1 : n flex] real*, и описание выполнится обычным путем. При этом на границы, содержащиеся в описании *row*, в данном случае просто не будет обращено никакого внимания. (Если же заменить в описании *row* на *[1 : n flex] real*, то оно станет синтаксически неверным.)

Казалось бы, роль индикантов и описаний вида сводится лишь к возможностям сокращения записи: вместо того чтобы несколько раз писать один и тот же сложный описатель, можно ввести описание вида с этим описателем в правой части и далее вместо него использовать индикант (играющий, таким образом, роль названия рассматриваемого класса объектов — значений данного вида). На самом же деле здесь появляются принципиально новые возможности в смысле спецификации новых видов, не являющихся окончательными порождениями *MODE* (и поэтому сразу же возникает задача соответствующего расширения множества видов). Поясним это на примере.

Рассмотрим следующие описания вида:

mode. language = struct (int *age*, ref language *father*)

(Это — не экзотика, пример дан самими авторами, и подобные вещи широко используются в языке.)

Какой вид специфицирует описатель *language* и специфицирует ли что-нибудь вообще?

Формально — нет, так как все порождения (в том числе и окончательные) всех понятий и метапонятий (в том числе и метапонятия *MODE*) представляют собой

конечные слова, поэтому никакое окончательное порождение *MODE* не проходит в качестве вида, специфицируемого описателем *language*.

(Действительно, такое слово *W* должно было бы удовлетворять равенству

*W = structured with integral field and reference to W field* что, очевидно невозможно.)

Тем не менее авторы языка считают, что подобные «рекурсивные» описатели специфицируют определенные виды, причем не дают по этому поводу никаких разъяснений. Это обстоятельство следует считать серьезной логической незавершенностью языка в той его форме, как он описан в [2].

Однако проведенные в связи с этим вопросом исследования показали, что множество видов можно так пополнить, что все рекурсивные описатели действительно будут специфицировать определенные виды (являющиеся не окончательными порождениями *MODE*, а элементами расширения), причем разные описатели будут специфицировать одинаковые виды в тех и только тех случаях, когда это так и должно быть по существу. Поэтому указанная выше незавершенность может быть снята простой ссылкой на такую возможность и указанием на то, что в качестве множества видов рассматривается не само множество окончательных порождений *MODE*, а его пополнение.

Интересующихся этим вопросом мы отсылаем к [5], здесь же больше не будем на этом останавливаться.

### § 13. Общее строение выражений и описаний

Мы разобрали основные частные случаи унитарных выражений и все унитарные описания (и те и другие можно определенным образом комбинировать, строя еще более крупные конструкции). Дадим теперь их общую синтаксическую иерархию (опуская детали, связанные с приведениями, см. § 1 гл. IV):

*unitary MOID clause: MOID tertiary; MOID confrontation.*  
*MOID tertiary: MOID secondary; MOID ADIC formula.*  
*MOID secondary: MOID primary; MOID selection;*  
*MOID generator.*

*MOID primary: MOID base; CLOSED MOID clause.*

В класс *confrontations* (сопоставлений) входят присваивания, ядра, а также еще два класса конструкций,

которые мы не рассматривали (один из этих классов будет рассмотрен в главе V).

Третье правило содержит тупики, так как выделения полей определены только для видов (нет понятия *void selection*), а генераторы — только для видов имен.

Метапонятие *CLOSED* имеет три порождения: *closed* (закрытый), *collateral* (параллельный) и *conditional* (условный).

Смысл последнего правила заключается в том, что любое выражение в скобках того или иного типа считается первичным. Далее мы разберем строение закрытых, параллельных и условных выражений.

Синтаксис унитарных описаний тривиален:

*unitary declaration: identity declaration;*  
*operation declaration;*  
*priority declaration;*  
*mode declaration.*

С описаниями мы сейчас покончим, разобрав последний их класс — *параллельное описание* (*collateral declaration*).

Параллельное описание состоит из нескольких (по крайней мере двух) унитарных описаний, отделенных друг от друга запятой. Например, конструкция

$\text{real } x = 3.14, \text{ int } n = 1$

представляет собой параллельное описание, состоящее из двух описаний идентичности.

Выполнение параллельного описания заключается в параллельном выполнении его компонент. Поэтому, например, результат выполнения описания

$\text{real } a = 3.14, \quad \text{real } b = a + 1$

(в расширенном языке  $\text{real } a = 3.14, b = a + 1$ ) будет неопределенным, так как он зависит от фактического порядка действий. Если же написать

$\text{real } a = 3.14; \quad \text{real } b = a + 1$

то мы получим два описания идентичности, выполняемые последовательно, и все будет в порядке.

Если несколько описаний можно объединить в одно параллельное, то, конечно, следует это сделать, так как это может дать выгоды в случае хорошей реализации. (Расширения подчеркивают желательность параллельных описаний, давая соответствующие возможности со-

кращений. Например, в вышеприведенной «параллельной» записи можно опустить второе вхождение `real`, в «последовательной» же — нельзя.)

Теперь должно стать ясным, почему в обозначениях процедур разрешается отделять друг от друга формальные параметры как запятой, так и точкой с запятой: при выполнении вызовов соответствующие описания идентичности разобьются на группы, разделенные точками с запятой, тогда как в одной группе описания (если их несколько) разделяются запятыми. В пределах одной такой группы описания будут выполняться параллельно, а сами эти группы — последовательно.

Обратимся теперь к общему строению выражений.

**З а к р ы т о е   в ы р а ж е н и е.** Закрытое выражение представляет собой заключенное в скобки *последовательное* (*serial*) выражение, и его выполнение состоит из выполнения этого последовательного выражения. (Скобки играют обычную синтаксическую роль — превращают крупную конструкцию в первичное выражение. Заметим, что здесь равно допускаются как круглые скобки, так и `begin` и `end`. Вообще, символы `begin` и `end` не играют особой роли и сохранены, по-видимому, просто для удобства программистов.)

Рассмотрим строение последовательного выражения. Последовательное выражение в АЛГОЛе-68 аналогично внутренности блока в АЛГОЛе-60 в том смысле, что оно является уровнем для областей действий имен и правил идентификации (идентификаторов, знаков операций и индикантов вида).

Уровнем в АЛГОЛе-68 считается именно последовательное, а не закрытое выражение (в отличие от АЛГОЛа-60, где уровнем локализации является блок). Так сделано потому, что последовательное выражение может занимать позиции между ограничителями, отличными от скобок (см. ниже о структуре условного выражения), сохраняя при этом роль уровня.

Так как в описаниях, входящих в состав последовательного выражения, могут описываться идентификаторы, знаки операций и индиканты вида, уже описанные на других уровнях, то при выполнении последовательного выражения следует застраховаться от всяческих неприятностей, связанных с совпадением наименований (могуших иметь место, например, при выполнении вызова, когда фактический параметр совпадает с идентификатором,

содержащимся в вызываемой рутине). Поэтому выполнение последовательного выражения всегда начинается с его *защиты*. Защита представляет собой совокупность действий, сводящихся к изменению всех идентификаторов, знаков операций и индикантов вида, описанных на данном уровне и встречающихся на уровне, внешнем по отношению к данному. Это, правда, громоздко, но зато надежно (защита последовательных выражений будет, конечно, реализовываться более «мягкими» методами, обеспечивающими устранение упомянутых неприятностей).

В дальнейшем мы не будем упоминать о защите, помня, однако, что с нее всегда начинается выполнение последовательного выражения.

Частный случай последовательного выражения — унитарное выражение. Тогда его выполнение состоит (не считая защиты) из выполнения этого унитарного выражения.

Перед унитарным выражением может стоять один или более *унитарных операторов* (*unitary statements*), разделенных точками с запятой и, возможно, помеченных.

Унитарный оператор определяется следующим образом: *unitary statement: strong unitary void clause*.

Слово *strong*, как мы увидим в главе IV при рассмотрении приведений, в данном случае разрешает помещать в позицию унитарного оператора любое унитарное выражение произвольного вида, например произвольное присваивание. В таком случае это выражение выполняется обычным путем, после чего перестает обладать своим значением. Например, после выполнения оператора  $x := 3.14$  идентификатор  $x$  будет обладать именем, ссылающимся на значение 3.14, сам же оператор — нет.

«Чистым» оператором (т. е. не требующим приведений) является, например, *void call*. Операторы могут помечаться так же, как в АЛГОЛе-60 (мы, правда, еще не рассматривали переходы к метке, но они имеются).

Выполнение всего такого последовательного выражения аналогично выполнению составного оператора в АЛГОЛе-60. Выполнение завершается, когда завершено выполнение последнего унитарного выражения, его значение (если оно есть) и есть значение всего выражения.

Примеры:

$$x := y; p(1); x + 1$$

$$x := 1; y := 0.1$$

В первом примере выполнится присваивание, вызов и формула  $x + 1$ ; значением всего выражения будет значение этой формулы.

Во втором примере последовательно выполнятся два присваивания. Значением выражения будет имя, которым обладает идентификатор  $y$ . В случае необходимости (если по контексту требуется, чтобы это последовательное выражение вырабатывало значение вида *real*) через это имя возможен доступ к значению 0.1, которое ему было присвоено. Для этого имя должно подвергнуться соответствующему приведению (см. гл. IV).

Пойдем дальше. Последовательное выражение может состоять из нескольких конструкций рассмотренных типов, отделенных друг от друга точкой, за которой следует метка и двоеточие. Например:

```
 $x := a + 1$ ; if  $p$  then go to second train fi;  
if  $x > 0$  then go to second train else  $x := 1 - x$  fi; false.  
second train:  $y := y + 1$ ; true
```

(Мы еще не рассматривали условных конструкций и переходов, но их смысл должен быть ясен.)

Здесь написано последовательное логическое выражение, состоящее из двух частей, разделенных точкой и меткой. Такие части называются *шлейфами* (*trains*).

Выполнение такого выражения начинается с выполнения первого шлейфа, но, как видно из примера, в процессе этого выполнения возможен переход к другому шлейфу (метка после точки обязательна, так как иначе не было бы доступа к началу шлейфа). Выполнение всего выражения завершается, когда завершится выполнение какого-либо его шлейфа, значение этого шлейфа и есть значение выражения.

Перед первым шлейфом (который, разумеется, может быть и единственным и состоять из одного унитарного выражения) может стоять одно или несколько описаний (унитарных или параллельных), отделенных друг от друга и от начала шлейфа точками с запятой. (Например, приведенное выражение могло начинаться с описания идентичности  $\text{real } a := 1$ ). Тогда выполнение выражения начинается с последовательного выполнения этих описаний.

Определяющие вхождения описываемых идентификаторов и индикантов идентифицируются только внутри данного последовательного выражения (но не внутри вложенного выражения, в котором они переописаны).

То же самое касается меток, так что переход извне на метку, расположенную внутри последовательного выражения, невозможен (так же, как в АЛГОЛе-60 нельзя перейти извне на метку, расположенную внутри блока).

Наконец, между каждыми двумя описаниями (а также перед первым из них) может помещаться один или более унитарных операторов, разделенных опять-таки точкой с запятой. Тогда последовательно вперемежку с описаниями выполняются и эти операторы. Например:

`int n; read(n); [1 : n] real x1, x2`

(пример дан в расширенном языке).

По смыслу эти промежуточные операторы должны нести вспомогательную нагрузку, подготавливая нужную информацию для выполнения описаний. Поэтому они считаются операторами одноразового действия, и метить их не разрешается.

При рассмотрении областей действия значений (см. § 7 гл. I) мы упоминали, что язык устроен таким образом, что значениями могут обладать только внешние объекты, лежащие внутри областей действий этих значений. Это достигается тем, что если область действия значения, полученного при выполнении последовательного выражения (с учетом приведений, см. гл. IV), не больше самого этого выражения, то значение последовательного выражения объявляется неопределенным.

**П а р а л л е л ь н о е     в ы р а ж е н и е.** Параллельное выражение, так же как закрытое, начинается с открывающей скобки и заканчивается закрывающей. Здесь также наряду с круглыми скобками допускаются скобки `begin` и `end`. Кроме того, для большей наглядности перед открывающей скобкой можно поместить символ `par` (*parallel symbol*). Внутри скобок помещается два или более унитарных выражений, разделенных запятыми. Выполнение параллельного выражения начинается с параллельного выполнения его компонент.

Если параллельное выражение занимает позицию оператора (т.е. является *collateral void clause*), то оно состоит из унитарных операторов, и выполнение его на этом завершается. В противном случае вид параллельного выражения начинается либо с *row of*, либо со *structured with*, и состоит оно соответственно либо из унитарных выражений того вида, который следует за этим *row of*, либо из унитарных выражений тех видов, которые

перечислены в качестве видов полей вслед за *structured with*. Во втором случае структурный вид предписывает также число и последовательность выражений.

В кратном случае определенным образом формируется новое кратное значение, в структурном — структурное. Мы не будем приводить соответствующую семантику, в которой опять фигурируют разные шаги, вычисления и сверки границ и т. п., а просто приведем примеры, в которых используются параллельные выражения.

[ ] *real*  $x := (1, 2, 3, 4, 5)$

[ , ] *real*  $y := ((1, 2), (3, 4))$

*struct* (*real re, im*)  $z := (0, 1)$  или *compl*  $z := (0, 1)$

*real*  $x, y$ ; *par begin*  $x := 0, y := 1$  *end*

(все примеры даны в расширенном языке).

Условное выражение. Как уже упоминалось, условное выражение в АЛГОЛе-68 обязательно заканчивается символом *fi*. Это дает возможность помещать после *then* и *else* произвольные последовательные выражения. Точно так же и после *if* помещается последовательное логическое выражение. Символ *else* и следующее за ним выражение в силу расширений могут отсутствовать, тогда это трактуется так, как если бы было написано *else skip*. В принципе же семантика условной конструкции такая же, как в АЛГОЛе-60.

Заметим, что вместо символов *if, then, else* и *fi* можно писать соответственно (, |, |, ).

Примеры:

*if overflow then go to exit fi*

$(x > 0 | x | 0)$

*if*  $x := 1; y := a < b$  *then* *real*  $z := a + b \times c; z \uparrow 3 + z \uparrow 2$   
*else* 0 *fi*

В третьем примере условие состоит из одного шлейфа. Завершающее его присваивание с помощью приведенных дает условию логическое значение, в зависимости от которого выполняется выражение, стоящее либо после *then*, либо после *else*. Выражение после *then* состоит из описания идентичности и завершающей формулы. После *else* стоит обозначение.

На этом мы закончим обзор основных конструкций АЛГОЛа-68 (программа — частный случай закрытого, параллельного или условного оператора) и займемся рассмотрением столь часто ранее упоминавшегося аппарата приведенных.

## ПРИВЕДЕНИЯ

## § 1. Смысл и общая схема аппарата приведений

Рассмотрим следующий текст на АЛГОЛе-60:

```
real x;    x: = y
```

Спрашивается, чем здесь может быть  $y$  (т.е. как может быть описан идентификатор  $y$ )?

Во-первых, конечно, вещественной переменной (т.е. с описанием **real**  $y$ ). Тогда значение переменной  $y$  (вещественное число) просто присвоится переменной  $x$ .

Во-вторых,  $y$  может быть целой переменной (т.е. с описанием **integer**  $y$ ). Тогда значение  $y$  (целое число) сначала переведется в эквивалентное вещественное (например, в 1.0 из 1), которое и присвоится  $x$ .

В-третьих,  $y$  может быть вещественной функцией без параметров (т.е. его описание будет начинаться с **real procedure**  $y$ ). Тогда сначала вычислится значение этой функции, и это значение (вещественное число) будет присвоено  $x$ .

Наконец,  $y$  может быть целой функцией без параметров (т.е. с началом описания **integer procedure**  $y$ ). Тогда вычислится значение этой функции, затем это целое значение переведется в эквивалентное вещественное, которое и будет присвоено  $x$ .

Мы видим, что если выражение типа **real** состоит даже только из одного идентификатора, то в силу имеющихся возможностей выполнения этот идентификатор совсем не обязан иметь тип **real** и может иметь разные виды в рамках этих возможностей. В АЛГОЛе-60 возможности такого варьирования сравнительно бедны и сводятся к помещению целых выражений в позиции вещественных (и наоборот) и функций без параметров в позиции переменных с очевидной семантикой. Поэтому этот вопрос не

вызвал в АЛГОЛе-60 никаких затруднений и особо не рассматривался.

Возможности же варьирования видов, которые предоставляет в принципе АЛГОЛ-68, оказались «джином, выпущенным из бутылки», и для его укрощения потребовалась специальная система понятий и синтаксических средств, которые в совокупности мы и будем называть аппаратом приведений.

Суть аппарата приведений заключается в следующем. Над значениями некоторых видов определяются специальные операции, называемые *приведениями* (*coercions*). Например, для имен определяется приведение, называемое *разыменованием* (*dereferencing*), результатом которого является значение, на которое имя ссылается. Для значений видов, порождаемых *procedure MOID* (т. е. процедур без параметров), определяется *распроцедурирование* (*deproceduring*), заключающееся в выполнении этих значений как закрытых выражений и рассмотрении результирующих значений, если таковые вырабатываются.

Набор приведений соответствует возможностям использования различных значений для получения других значений (переход от имени к значению, на которое оно ссылается, выполнение процедуры без параметров и др.). Далее мы рассмотрим все типы приведений.

С приведениями связаны два процесса. Один из них осуществляется во время чтения (синтаксического анализа) программы и заключается в построении цепочки порождаемых одно другим понятий, ведущей от выражения того вида  $M$ , который требуется по контексту в данной позиции, к объекту другого вида  $M_1$ , фактически помещенному в этой позиции. Найденная цепочка определяет второй процесс, осуществляемый во время выполнения программы. Он заключается в преобразовании значения вида  $M_1$  в значение вида  $M$ .

Первый процесс, заслуживающий наименования *приведения видов*, является вспомогательным. Собственно приведение — *приведение значений* — осуществляется во втором процессе.

Мы не будем излагать всех синтаксических деталей, обслуживающих аппарат приведений. Рассмотрим, однако, пример, который наряду с примерами, приводимыми ниже, может помочь читателю уяснить, как этот аппарат работает. Пусть контекст требует, чтобы в некоторой позиции программы было помещено порождение

понятия *strong row of real base*, а фактически эту позицию занимает идентификатор, который согласно его описанию является порождением понятия *reference to procedure real mode identifier*. Здесь вид *M* — это *row of real*, а вид *M<sub>1</sub>* — *reference to procedure real*. Синтаксис языка таков, что от первого понятия ко второму ведет только одна цепочка понятий, а именно:

*strong row of real base:*

*strongly rowed to row of real base:*

*strongly deprocedured to real base:*

*strongly dereferenced to procedure real base:*

*reference to procedure real base:*

*reference to procedure real mode identifier.*

Эта цепочка предписывает, что значение, которым обладает идентификатор (имя процедуры без параметров, вырабатывающей вещественное значение), разыменовывается (находится рутина, на которую ссылается это имя), распроцедурируется (эта рутинa выполняется) и векторизуется (формируется одномерный массив, единственным элементом которого является значение, выработанное процедурой). Полученный массив принимается за значение базы.

После того, как приведения определены, заманчиво поступить следующим образом: объявить, что в позициях выражений любого вида могут стоять не только собственно выражения этого вида (т. е. такие, в результате обычного выполнения которых получаются значения этого вида), но и выражения всех тех видов, от значений которых ведет цепочка приведений (сколь угодно длинная) к значению нужного вида. Тогда после собственно выполнения этих выражений они будут определенным образом «довыполняться» с помощью операций приведений. (При этом составные части выражений снова могут быть приводящимися и т. д.)

Эту схему синтаксически осуществить сравнительно нетрудно (по отношению к тому, как сделано на самом деле), но, к сожалению, в такой форме она не проходит по крайней мере по двум причинам.

Во-первых, вообще говоря, можно потерять информацию о том, какого же вида значение мы хотим в конце концов получить, т. е., какого вида выражение представляет собой конкретная конструкция. Рассмотрим, например, такой текст (в расширенном языке):

**ref real *xx*;    real *x*;    *xx*: = *x***

Не будь приведений, все трактовалось бы однозначно: имя, которым обладает *x*, присвоилось бы имени (двойному), обладаемому *xx*. Но если мы допустим такую свободу, о которой говорилось выше, то почему бы не трак-

товать данное присваивание еще и так: разыменовать и имя *xx* (получив значение вида *reference to real*), и имя *x* (получив вещественное значение) и присвоить первому значению второе? Таким образом, в некоторых позициях следует ограничить свободу приведений, чтобы сделать эти позиции ответственными за информацию в виде рассматриваемого выражения.

Во-вторых, цепочек приведений, ведущих от полученного вида к требуемому, может оказаться более одной, и продвижение по разным цепочкам может дать разные эффекты (в § 3 будут приведены соответствующие примеры). Поэтому следует позаботиться об однозначности и с этой целью тоже соответствующим образом ограничить приведения в определенных позициях.

Все это привело к значительному усложнению синтаксиса АЛГОЛа-68. Более того, в процессе работы над языком выявлялись различные примеры неоднозначностей, что вызывало дальнейшие усложнения (и авторы языка не приводят никакого обоснования того, что эта «отладка» завершена и впоследствии не обнаружится что-либо неучтенное).

Мы не будем здесь вникать во все тонкости, связанные с приведениями, рассмотрим только общую схему.

Все позиции, занимаемые выражениями, подразделяются на несколько *сортов*. Каждому сорту ставится в соответствие определенное множество допустимых цепочек приведений, и занимать позицию разрешается только таким выражениям, от значения которых к значению нужного вида ведет допустимая цепочка.

С каждым сортом связывается некоторое слово, порождаемое метапонятием *SORT*. Вот какие различаются сорта:

*SORT<sub>1</sub> strong; firm; weak; soft.*

(Таким образом, различаются *сильные, твердые, слабые и мягкие* позиции. В сильных позициях допустимыми являются все возможные цепочки, остальные три сорта последовательно накладывают все большие ограничения.)

Когда выражение занимает позицию какого-либо сорта, этот сорт фигурирует в правилах синтаксиса. (Например, источник присваивания занимает сильную позицию, а его назначение — мягкую, см. § 6 гл. III.)

Теперь следует задать возможности для каждого сорта позиций. Это делается следующим образом.

В правилах, задающих общее строение крупных конструкций, и слева и справа фигурирует метапонятие *SORTETY*. В частности, это имеет место во всех правилах, относящихся к общему синтаксису унитарных выражений (см. § 13 гл. III). Например, последнее из этих правил на самом деле таково:

*SORTETY MOID primary: SORTETY MOID base;  
SORTETY CLOSED MOID clause.*

Метапонятие *SORTETY* определяется следующим образом:

*SORTETY: SORT; EMPTY.*

Таким образом, правила, приведенные в § 13 гл. III (и вообще, правила, получаемые удалением вхождений *SORTETY*), являются частным случаем полных правил, соответствующим пустому порождению *SORTETY*. В этом случае порождаются выражения только данного вида. В случае же какого-либо сорта (т.е. порождения *SORT*) правила должны порождать все те выражения, которым разрешено занимать для данного вида позицию этого сорта (в том числе, конечно, и сами выражения данного вида). Следовательно, все прямые порождения понятий, в состав которых входят сорта, тоже должны быть понятиями и иметь порождения. Так оно и есть на самом деле. Например, одно из правил, определяющих закрытое выражение через последовательное (см. § 18 гл. III), таково:

*SORTETY closed MOID clause: open symbol,  
SORTETY serial MOID clause,  
close symbol.*

В свою очередь в правилах, определяющих общее строение последовательных выражений, снова слева и справа фигурирует *SORTETY* и т. д. Однако ясно, что такое перенесение «ответственности» за то, что же в конце концов будет порождаться, должно где-то закончиться, т.е. должны быть конструкции, в синтаксисе которых явно указаны возможности для каждого сорта позиции и при выполнении которых осуществляются приведения. Такие конструкции действительно имеются и называются *приводимыми* (*coercends*). К приводимым относятся сопоставления (в частности, присваивания и ядра), формулы, генераторы, выделения полей и базы. Для всех

приводимых формулируются специальные синтаксис и семантика для каждого сорта позиции и типа приведения, и общая схема сводится к следующему:

а) для каждого сорта позиции, категории приводимого и вида, связанного с этим приводимым, синтаксис языка определяет множество допустимых цепочек понятий, ведущих к понятиям, уже лишенным сорта; при этом для всех таких цепочек виды, связанные с их заключительными понятиями, различны;

б) для некоторых переходов в такой цепочке от понятия к его прямому порождению семантика языка определяет операции приведения значения, причем последовательность этих операций при выполнении программы соответствует движению от конца цепочки к ее началу.

Рассмотрим пример. Знак  $+$  в стандартных описаниях определен для вещественных операндов. При синтаксическом анализе формулы  $x + 0.5$ , где по описанию  $x$  имеет вид *reference to real*, необходимо построить цепочку, ведущую от понятия *real operand* (детали, связанные с приоритетами знаков операций мы сейчас опускаем) к понятию *reference to real mode identifier*. Начальное звено этой цепочки определяется однозначно:

*real operand:*  
*firm real secondary*

Тем самым определяется сорт (*firm* — твердый) позиции. Часть цепочки, ведущую от понятия *firm real secondary* к понятию *firm real base*, опускаем. Существенно лишь то, что сорт *firm* передается по всем звеньям этой цепочки. Понятие *firm real base* — это уже приводимое. Здесь и надо постараться найти цепочку, ведущую к понятию вида *reference to real*. Она оказывается такой:

*firm real base:*  
*firmly dereferenced to real base:*  
*reference to real base.*

Среди многочисленных вариантов первого звена выбран такой, который связан с необходимым в данном случае приведением (разыменованием). Второе звено предписывает выполнение разыменования во время выполнения программы

Более сложный пример цепочки приведений был дан выше.

В качестве еще одного примера рассмотрим запись

$xx := 1.5,$

где идентификатор  $xx$  по описанию имеет вид

*reference to reference to real*

Если вид присваивания — *reference to M*, то надо попытаться найти цепочки, ведущие от

*reference to M destination*

к

*reference to reference to real mode identifier*

и от *M source* к *real denotation*. Первые звенья этих цепочек определяются однозначно синтаксисом назначения и источника:

*reference to M destination:*

*soft reference to M tertiary*

и

*M source:*

*strong M unit*

Однако при любом выборе  $M$  (выбирать можно только между *real* и *reference to real*) дальнейшее построение либо той, либо другой цепочки оказывается невозможным. Приведения значения вида *real* к значению вида *reference to real* (если принять, что  $M$  — это *reference to real*) вообще не существует. Приведение же значения вида *reference to reference to M* к виду *reference to M* (если считать, что  $M$  обозначает *real*) существует (разыменование), но в мягких (*soft*) позициях синтаксически не допускается, так как в языке нет соответствующего правила (которое должно было бы выглядеть так: *soft COERCEND: softly dereferenced to COERCEND*).

Здесь может выручить ядро, например: (*ref real : xx*)  
 $i = 1$ , так как позиция после двоеточия — сильная (*strong*), разрешающая разыменование, требуемое описателем *ref real*.

В синтаксисе АЛГОЛа-68 предпринята попытка вылавливать буквально крохи информации о возможных видах выражений, и всюду, где этой информации оказывается уже достаточно, позиция объявляется сильной (т. е. разрешаются все приведения, так как ясно, к чему приводить). Но следует заметить, что синтаксические ухищрения, которыми это достигается, могут вызвать у читателя «бессильную злость непонимания».

Мы ограничимся здесь перечислением типов приведений, их семантикой, т.е. выполнением операций приведений, а также упомянем, какие цепочки приведений допускаются для каждого сорта позиций. Приведем также несколько примеров. Рассмотрение же всех синтаксических и семантических деталей, связанных с приведениями, увело бы нас слишком далеко.

## § 2. Типы приведений

Предположим, что приводимое занимает сильную позицию (т.е. в соответствующем правиле синтаксиса понятие, порождающее, в частности, это приводимое, начинается со слова *strong*, как, например, в синтаксисе источника присваивания). Пусть далее его вид не тот, какой требуется в этой позиции. Рассмотрим, в каких случаях такая ситуация допустима (и узаконивается синтаксисом), т.е. какие вообще бывают приведения и какие действия они определяют после того, как приводимое обычным образом выполнится в качестве выражения своего вида.

1) Если полученное значение — имя (например, выполнялось присваивание или генератор), то может быть произведено разыменовывание, результатом которого будет значение, на которое это имя ссылается. После разыменовывания позиция, занимаемая приводимым, по синтаксису продолжает оставаться сильной, так что можно двигаться дальше по цепочке ссылок.

2) Если полученное значение — рутина одного из видов, порождаемых *procedure MOID* (например, в случае выполнения идентификатора такого вида), то может быть произведено распроедуривание путем вызова этой рутины, ее выполнения как закрытого выражения и восстановления приводимого на своем месте. В результате либо получится какое-то значение, либо (в случае вида *procedure void*) ничего не получится (но, может быть, *void* — это как раз то, что нужно). После распроедуривания позиция продолжает оставаться сильной, так что если полученное значение имеет один из рассмотренных видов, то снова может быть произведено либо разыменовывание, либо распроедуривание.

Таким образом, в сильной позиции допустимой является, в частности, любая цепочка приведений, состоящая из разыменовываний и распроедуриваний, и если от исходного значения к значению нужного вида ведет

такая цепочка, то все в порядке, и нужное значение получится после выполнения требуемого числа указанных действий (возможно, попеременно).

3) Если полученное значение — целое число (какой-либо длины), то может быть произведено приведение, называемое *расширением* (*widening*), которое состоит в замене этого целого эквивалентным вещественным числом (той же длины). Вещественное значение может быть расширено до эквивалентного ему комплексного, т. е. до структурного с двумя вещественными полями, первое из которых есть расширяемое вещественное, а второе — «вещественный» нуль той же длины (см. § 12 гл. III).

Заметим, что приведения, осуществляющего переход от вещественного значения к целому, нет, поэтому в позиции целого выражения вещественное стоять не может, и в программе нужно явно указывать, какая из имеющихся стандартных операций (округление до ближайшего целого — *round* или взятие целой части — *entier*) должна быть применена для выработки нужного целого значения.

4) К полученному значению любого вида может быть применено приведение, называемое *векторизацией* (*rowing*). Упрощенно выполнение векторизации заключается в следующем.

Если полученное значение — не кратное и не имя, то результатом векторизации является кратное значение с паспортом, состоящим из начала отсчета 1 и пятерки (1, 1, 1, 1, 1) и с единственным элементом — старым значением, т. е. просто не кратное значение начинает рассматриваться как одномерный массив, состоящий из одного этого элемента. (Вот почему литерное обозначение — *character denotation* — выделялось особо (см. § 2 гл. III): его можно помещать как в позиции вида *character*, так и в сильной позиции вида *row of character*. В последнем случае работает векторизация.)

Если полученное значение — кратное, то увеличивается на единицу его размерность (без изменения состава элементов): начинает рассматриваться кратное значение, получаемое добавлением к паспорту старого значения еще одной пятерки (1, 1,  $d$ , 1, 1), помещаемой перед старой первой пятеркой. Шаг  $d$  при этом вычисляется с помощью границ и шага этой пятерки по формуле

$$d = (u_1 - l_1) \times d_1$$

Наконец, если старое значение — имя, то векторизуется значение, на которое это имя ссылается (любое, в том числе и имя), и на это значение начинает ссылаться имя нужного вида (так как вид значения изменился, то требуется и другое имя).

Перед векторизацией могут выполняться любые из рассмотренных приведений, после векторизации — только снова векторизация.

Примеры конструкций, при выполнении которых будет произведена векторизация:

```
[1 : 1] real x = 3.14
[1 : 1, 1 : 1] real y := 3.14
[1 : 1] compl z := 3.14
```

Во втором примере векторизация будет произведена дважды. В третьем примере перед векторизацией будет произведено расширение.

5) Следующий тип приведения, который мы рассмотрим, называется *опустошением* (*voiding*). Это приведение заключается в том, что полученное значение забывается, а приводимое перестает им обладать и лишается вида (и, следовательно, может быть оператором, т.е. *void clause*).

Условия, при которых может быть произведено опустошение, мы сформулируем несколько упрощенно по сравнению с [2]. (Смысл опустошения, очевидно, заключается в том, чтобы разрешить выражениям различных видов занимать позиции операторов. Однако для некоторых видов это заведомо не имеет смысла, так как действие таких операторов будет эквивалентно пустому. Поэтому в отдельных случаях синтаксисом предусмотрено запрещение опустошения, и, следовательно, запрещение помещения выражений определенных видов в позиции операторов. Здесь, однако, для нас это несущественно, и мы для простоты разрешим опустошение во всех таких случаях.)

Опустошение может быть произведено, если

а) выполнявшееся приводимое — сопоставление (в частности, присваивание, см. § 13 гл. III);

б) вид полученного значения не порождается *procedure MOID* и не может быть к такому виду приведен.

Таким образом, в нашей трактовке любое унитарное выражение любого вида может занимать сильную

позицию унитарного оператора. Действительно, докажем это утверждение для любых приводимых (этого будет достаточно, так как по синтаксису к ним все сводится).

Если приводимое — сопоставление (в частности, присваивание любого вида), то в силу а) опустошение возможно, и «довыполнение» завершено.

В противном случае могут быть следующие два варианта:

1) Вид полученного значения не порождается *procedure MOID* и не может быть к такому виду приведен; тогда опустошение возможно в силу б).

2) Вид полученного значения или порождается *procedure MOID* или приводится к такому виду. Тогда опустошение не допускается, но такая ситуация означает, что возможны распроедуривания и, быть может, разыменования. А тогда эти приведения и будут выполняться до тех пор, пока опустошение не окажется допустимым.

Например, если идентификатор  $f$  вида *reference to procedure real*, то оператор  $f$  выполнится так:

а)  $f$  выполнится как идентификатор, после чего он будет обладать именем рутины;

б) будет произведено разыменование, после чего будет рассматриваться сама рутина;

в) будет произведено распроедуривание, т.е. выполнение этой рутины как закрытого выражения;

г) произойдет опустошение (т.е. забудется полученное вещественное значение), и выполнение оператора закончится.

В случае сопоставлений (в частности, присваиваний) опустошение всегда происходит сразу (так предписывает синтаксис). Это вызвано страховкой от некоторых нежелательных эффектов, далее мы приведем пример.

Итак, допустимые цепочки рассмотренных произведений в сильных позициях могут иметь следующее строение. Начинаться они могут с произвольного (возможно, нулевого) числа разыменований и распроедуриваний, идущих попеременно. Если на этом цепочка не заканчивается, то дальнейший ее участок может состоять либо из одного опустошения, либо из произвольного числа (возможно, нулевого) векторизаций, перед которыми, возможно, имеются одно или два расширения. (По синтаксису, если требуется приведение к *void*, то опустошение производится сразу, как только оно окажется возможным, поэтому опустошение не может следовать ни после расширения,

ни после векторизации. Это и естественно: незачем расширять или векторизовывать значение, если оно потом все равно будет утеряно. Распроцедуривание же имеет смысл: будет выполняться какая-то рутина, а ее выполнение может сопровождаться побочными эффектами, ради которых и принята такая схема. Разыменования тоже возможны, если после них будут распроцедуривания.)

Таким образом, если от вида  $M_1$  к виду  $M_2$  ведет цепочка такого строения, то сильную позицию вида  $M_2$  может занимать выражение вида  $M_1$ .

Рассмотрим еще две возможности несоответствия видов.

1) В сильных позициях видов, порождаемых *procedure MOID*, допустимыми являются не только конструкции, приводящиеся к этому виду описанными выше способами, но также и допустимые для сильных *MOID*-позиций. В таком случае не происходит обычного выполнения конструкции, а она рассматривается как обозначение процедуры без параметров: формируется рутина, которая и становится значением этой конструкции (рутина формируется так: вначале ставится *MOID declarer* и двоеточие, после чего получается *MOID cast*, и все заключается в скобки, образуя *closed MOID clause*). Эту рутину можно сделать значением идентификатора и затем вызывать ее через распроцедуривание. (Поэтому специальное понятие обозначения процедуры без параметров отсутствует.)

Такая операция называется *запроцедуриванием* (*proceduring*).

Примеры

а) `real x := 0.1; proc real p = x; p = 0.5`

Здесь написано последовательное логическое выражение. При выполнении второго описания идентичности произойдет запроцедуривание (позиция фактического параметра — сильная) и идентификатор  $p$  будет обладать значением

(real : x)

При выполнении идентификатора  $p$  в формуле  $p = 0.5$  произойдет распроцедуривание, т.е. выполнится ядро

real : x

При этом произойдет разыменование имени  $x$  (это диктуется описателем *real*) и получится вещественное

значение 0.1. Значение логической формулы (а с ней и всего логического выражения) будет, таким образом *false*.

б) `proc proc real p = 3.14`

Такая конструкция допустима, так как вид *real* допустим для *procedure real* (через запроцедурирование), а с ним и для *procedure procedure real* (через повторное запроцедурирование).

Идентификатор *p* будет обладать значением

`(proc real : 3.14)`

При выполнении этой рутины снова произойдет запроцедурирование и получится значение

`(real : 3.14)`

При выполнении этой рутины, наконец, получится вещественное значение.

Такие действия будут произведены, например, при выполнении следующего текста:

`proc real q = p; real x: = q`

в) `proc p; p: = go to stop`

По смыслу ясно, что посредством присваивания мы хотим только присвоить имени *p* рутину (`: go to stop`), но не выполнять ее. Поэтому в случае присваивания (и других сопоставлений, так как для них возможны аналогичные ситуации) и не разрешается перед опустошением производить разыменования и распроцедурирования. В противном случае в нашем примере можно было бы после «собственно» выполнения присваивания (в процессе которого происходит запроцедурирование) вместо немедленного опустошения произвести разыменование с последующим распроцедурированием, т. е. выполнением присвоенной рутины.

2) Сильную позицию выражения любого вида могут занимать две конструкции: *пропуск* (*skip*) и *переход* (*jump*).

Пропуск состоит из одного символа *skip*. Если пропуск занимает позицию оператора, то его выполнение не влечет никаких действий (так что в этих позициях пропуск играет роль пустого оператора и может служить для помещения метки). В противном случае, как уже говорилось, рассматривается некоторое (неизвестно, ка-

кое) значение нужного вида, и оно становится значением пропуска. Областью действия такого значения всегда (даже в случае имени) считается вся программа. Таким образом, употребление пропусков в позициях видов имен дает эффект, похожий на даваемый глобальными генераторами. Однако с пропусками следует обращаться осторожно, так как, в отличие от генераторов, здесь не говорится, что значение пропуска — новое, отличное от всех других. Поэтому при попытке присвоить что-нибудь имени, порожденному с помощью пропуска, можно «испортить» какое-либо конкретное используемое в программе имя, так как оно было взято реализацией в качестве значения пропуска. (Более того, при реализации произвольное имя может трактоваться для простоты как «адрес любой, все равно какой ячейки», например, одной из используемых рабочих ячеек или даже ячейки, содержащей команду. Тогда при присваивании такому имени есть риск взорвать всю систему.)

Переход — это привычная по АЛГОЛу-60 конструкция, состоящая из *go to* и идентификатора метки, причем *go to* можно не писать. Метки не имеют вида, и никаких именующих выражений в языке нет, но они легко моделируются через процедуры. Вспомним пример, приведенный при рассмотрении описаний идентичности (см. § 10 гл. III):

[ ] *proc* *s* = (*go to* *l1*, *go to* *l2*, *go to* *l3*)

где *l1*, *l2*, *l3* — идентификаторы меток.

Здесь справа стоит параллельное выражение вида *row of procedure void*, поэтому его компоненты должны быть вида *procedure void*. Позиции компонент сильны, так что их могут занимать переходы. Так как в записи переходов можно опускать *go to*, то наш пример может быть записан короче:

[ ] *proc* *s* = (*l1*, *l2*, *l3*)

В случае, когда переход стоит в позиции любого из видов *procedure MOID*, то он запроцедурируется. Поэтому в результате выполнения данного описания идентичности идентификатор *s* будет обладать кратным значением, элементы которого — три рутины:

(: *l1*), (: *l2*), (: *l3*).

Оператор  $\bar{s}[i]$  выполняется сначала как вырезка (получится рутина), затем произойдет распроцедурирование (опустошать нельзя) и выполнится переход на одну из меток.

Но заметим, что если, например, переход  $l$  занимает позицию вида *procedure procedure void*, то его выполнение даст рутину (*прос : l*), выполнение которой даст (*: l*), и только выполнение этой рутины осуществит сам переход.

### § 3. Ограничения приведений

Рассмотрим, какие ограничения на приведения накладывают не сильные позиции.

В твердых позициях запрещается помещать пропуски и переходы, запрещается производить расширение и векторизацию, а также не может быть произведено опустошение.

Опустошение не в сильных позициях заведомо невозможно, так как все эти позиции требуют приведения к виду, а не к *void* (все *void*-позиции сильные).

Запрещение расширения и векторизации связано с однозначностью приведений. Рассмотрим, например, формулы  $a \times b$  и  $i + j$ . Предположим, что операция умножения определена для чисел и матриц, а операция сложения — для целых и вещественных чисел. При этом разные «умножения» и «сложения» действительно совершенно различны и никак не связаны. Пусть далее  $a$  и  $b$  — идентификаторы вида *real*, а  $i$  и  $j$  — вида *integral*. Тогда, если в позициях операндов разрешить векторизацию, то неясно, как выполнять  $a \times b$ : перемножить ли числа или, произведя двойную векторизацию, перемножить матрицы (ни о какой эквивалентности результатов таких процессов выполнения, вообще говоря, не может идти речи, так как операциям можно давать произвольные определения). Точно так же, если разрешить расширение, то неоднозначна трактовка  $i + j$ : неясно, применить ли «целую» операцию к целым числам, или, расширив их, применить «вещественную» операцию.

Поэтому позиция операнда в формуле объявляется твердой, чем подобные неоднозначности устраняются.

Однако формулы все равно доставляют еще много хлопот, так как, например, разыменование в твердой позиции допускается (запретить его бессмысленно, потому что тогда нельзя будет писать элементарнейшие

вещи, например, вместо  $x := x + 1$  придется писать  $x := (\text{real} : x) + 1$ ). Поэтому трактовка формулы  $x + y$ , если  $+$  определен и для чисел и для имен, неоднозначна. (Заметим, что все эти трудности вызваны только возможностью описывать разные операции с одним знаком, так что за эту «математизацию» языка приходится платить довольно дорого.) Авторы выходят из этого положения с помощью «крайних мер»: вводят помимо синтаксиса еще серию так называемых *контекстных условий*, содержащих различные ограничения и запрещения. Одно из контекстных условий запрещает давать один знак операциям, виды операндов которых могут вызывать неоднозначности в твердых позициях. Например, если  $+$  определен для операндов вида *real*, то запрещается его параллельно определять для операндов вида *reference to real*.

Запрещение пропусков и переходов в твердых позициях также вызвано страховкой от неоднозначностей. Например, трактовка формулы  $1 + \text{skip}$  неоднозначна, если  $+$  определен дважды с одним и тем же видом первого операнда (например, *integral*) и разными видами второго (так оно и есть на самом деле: согласно стандартным описаниям второй операнд здесь может быть как вида *integral*, так и вида *real*).

Таким образом, допустимые цепочки рассмотренных нами приведений в твердой позиции состоят из разыменований и распроцедуриваний, и, кроме того, приводимое, занимающее твердую позицию, может быть запроцедурено. (В следующей главе мы рассмотрим еще один тип приведения. Это приведение допускается в сильных и твердых позициях.)

Рассмотрим теперь слабые позиции.

Слабыми объявляются позиции выражения перед индексными скобками в вырезке и после *of* в выделении поля. В этих позициях, конечно, не нужно ни опустошения (так как приводить нужно к виду), ни векторизации, ни расширения. Поэтому эти приведения в слабых позициях не допускаются. Не может быть также и запроцедуривания приводимых, стоящих в слабой позиции, так как оно производится только в случае видов, порожденных *procedure MOID*.

Таким образом, в слабых позициях допускаются только разыменования и распроцедуривания, да и то с одной оговоркой, принципиально, впрочем, несущественной.

Наконец, в мягкой позиции запрещается и разыменование, и разрешается только распроедуривание. Мягкой является, например, позиция назначения присваивания. Это дает однозначную трактовку такого, например, текста:

**ref real  $xx$ ,      real  $x$ : = 0;       $xx$ : =  $x$**

Так как здесь при выполнении присваивания нельзя разыменовывать имя  $xx$ , то именно этому имени будет присвоено имя  $x$ .

На этом мы закончим рассмотрение аппарата приведений. Более подробное рассмотрение потребовало бы либо слишком много места, либо более формального стиля изложения, близкого к стилю [2]. И то и другое для наших целей нежелательно.

## ОБЪЕДИНЕННЫЕ ВИДЫ И ОТНОШЕНИЯ СОГЛАСОВАНИЯ

### § 1. Объединенные виды

В АЛГОЛе-60 возможность не снабжать формальные параметры спецификациями и неполнота спецификаций в случае массивов и процедур вызвали к жизни заманчивую, но расплывчатую концепцию «динамического» вида. Обсуждалась даже допустимость текстов, подобных, например, следующему:

```
real procedure f(a, x); f: = if a = 1 then x[i] else x[i, i]
```

Дескать в зависимости от того, каким будет первый параметр, второй параметр будет либо одномерным, либо двумерным массивом и в динамике вычисления будут осмысленными.

Такое рассуждение неудовлетворительно, потому что после замены формального параметра  $x$  фактическим конструкцией станет бесспорно недопустимой (впрочем, можно идти дальше и обсуждать допустимость текстов типа

```
array y[1:10]; if 0 = 1 then x: = y[i, i]
```

но это уже очевидная бессмыслица). Но ясно, что иметь возможность какого-либо варьирования вида одной и той же величины (в терминах АЛГОЛа-68 — вида значения, обладаемого внешним объектом) может оказаться полезным.

В АЛГОЛе-68 в этом направлении предприняты некоторые шаги, но предоставляемые возможности четко регламентируются, не оставляя места для разноречивых толкований.

Суть дела заключается в следующем. Множество видов расширяется новыми видами, называемыми *объединенными*.

Мы дадим упрощенную трактовку строения объединенных видов и соответствующих им описателей, а именно

ту, которая принята в [1]. (В [2] синтаксис описателей объединенных видов сильно усложнен с тем, чтобы разные описатели, соответствующие одинаковым по смыслу объединениям, специфицировали бы одинаковые виды. Здесь для нас это несущественно.)

Объединенные виды порождаются метапонятием *UNITED*.

*UNITED: union of MODES mode.*

*MODES: MODE; MODES and MODE.*

В метapравило, определяющее *MODE*, добавляется *UNITED*:

*MODE: TYPE; STOWED; UNITED.*

Остальные метapравила, связанные с видами, остаются без изменений.

Таким образом, не объединенные виды могут содержать в своем составе объединенные (т.е. слова, являющиеся видами и не начинающиеся с *union of*, могут иметь вхождения слов, являющихся объединенными видами, например: *reference to union of real and boolean mode*).

Описатели, специфицирующие объединенные виды, состоят из символа *union* и заключенных в скобки описателей (всегда виртуальных) тех видов, «объединение» которых представляет объединенный вид. Эти описатели отделяются друг от друга запятой.

Например:

*union (real, union (int, bool), union (real, int))*  
*union (int, real, bool)*

(по смыслу работы с объединенными видами виды, специфицируемые этими описателями, должны быть одинаковыми: фактически объединяются виды *integral*, *real* и *boolean*. Согласно [2] так и будет, в нашей же трактовке специфицируемые виды — разные, но здесь нам это не важно).

Смысл введения объединенных видов будет ясен, если знать, какими значениями могут обладать выражения этих видов. Эти выражения могут обладать значением любого вида, входящего в состав объединенного. Но виды значений по определению никогда не бывают объединенными, поэтому если «внутренний» вид снова объединенный, то вид обладаемого значения — любой из входящих в его состав и т.д. Например, идентификаторы видов, спе-

цифицируемых вышеприведенными описателями, могут обладать целыми, вещественными и логическими значениями (поэтому и утверждалось, что эти виды по смыслу одинаковы).

Возникает вопрос: какие значения имеют вид, который не будучи сам объединенным, содержит объединенный вид в своем составе? Например, каким значением обладает идентификатор  $x$  после выполнения следующего описания идентичности:

`union (int, bool) x: = 1`

Вспомним, что значение можно рассматривать как «учетную карточку», на которой написаны, во-первых, его вид и, во-вторых, само значение. В данном случае на карточке имени, которым обладает идентификатор  $x$ , будет написан вид *reference to union of integral and boolean mode*, ссылаться же это имя будет на значение вида *integral* — единицу. Но написанный вид сделает возможным, например, присваивание  $x := \text{true}$ , что было бы недопустимо в случае вида *reference to integral*.

Таким образом, если в составе вида значения содержится объединенный вид, то само это значение сформировано из конкретных значений, но запись объединенного вида на карточке дает возможность варьирования видов этих значений в рамках объединения. Между прочим, появляется любопытная возможность формирования «разношерстных» массивов, например:

`[1 : 4] union (int, bool) ib: = (1, true, 0, false)`

Синтаксические возможности для получения значений выражений объединенных видов задаются с помощью еще одного приведения — *объединения (uniting)*. Объединение не связывается ни с какими операциями над значением во время выполнения приводимого. Оно лишь позволяет (при определенных условиях) поместить приводимое некоторого конкретного вида  $M$  в позицию, где по контексту должно размещаться выражение вида, получающегося объединением  $M$  с другими видами.

Объединение допускается в сильных и твердых позициях (с одной оговоркой, связанной с однозначностью приведений, далее мы приведем пример). Поэтому множество допустимых цепочек приведений в этих позициях

пополняется цепочками, в состав которых входят объединения.

В частности, в вышеприведенном примере позиции компонент параллельного выражения — сильные, поэтому возможно объединение от видов *integral* и *boolean* до вида

*union of integral and boolean mode*

Рассмотрим такой текст:

**proc ref int  $p = x := 1$ ; union (int, bool)  $ib := p$**

Здесь при выполнении первого описания произойдет запроцедурирование присваивания  $x := 1$ , и  $p$  будет обладать значением

(ref int:  $x := 1$ )

При выполнении второго описания требуется приведение к виду

*union of integral and boolean mode*

Поэтому выполнение будет происходить следующим образом:

- 1)  $p$  выполнится как идентификатор и будет обладать той же рутиной, что и его определяющее вхождение;
- 2) произойдет распроцедурирование, т. е. выполнится ядро

ref int:  $x := 1$

в результате чего будет получено имя, ссылающееся на единицу;

- 3) произойдет разыменование, после чего будет рассматриваться сама единица;

- 4) создастся имя вида *reference to union of integral and boolean mode*, ссылающееся на единицу, и идентификатор  $ib$  начнет обладать этим именем.

Рассмотрим теперь следующее описание:

union (int, real)  $ir := 1$

Для того чтобы избежать в подобных ситуациях неоднозначности приведений, синтаксис разрешает производить объединение только после цепочек приведений, допустимых в твердых позициях. В частности, объединение не разрешается производить после расширения, так как расширение в твердых позициях запрещено. Поэтому

трактовка данного описания однозначна: единица не может быть расширена (с последующим объединением) и может быть только сразу объединена до требуемого вида *union of integral and real mode*. Аналогичный пример можно привести и с векторизацией.

Но рассмотрим такое описание:

**union (real, ref real) rr: = x**

Спрашивается, чем будет обладать *rr* после выполнения этого описания: именем, которым обладает *x* (сразу объединение) или вещественным значением, на которое это имя ссылается (сначала разыменованное, потом объединение)?

Здесь, таким образом, снова кроются неоднозначности приведений. Из этого положения авторы АЛГОЛа-68 выходят уже не синтаксическими средствами, а с помощью контекстного условия, запрещающего подобные ситуации (следует, впрочем, отметить, что все контекстные условия формализованы, так что их соблюдение может быть проверено автоматически).

Заметим, что приведения, обратного объединению — *разъединения* (*deuniting*), — нет, и выражения объединенных видов не могут занимать позиции объединяемых видов. Отсутствие такой возможности вызвано страховкой от ситуации, когда в данный момент значение «объединенного» выражения не того вида, который нужен.

## § 2. Отношения согласования

В связи с тем, что выражения объединенных видов могут в динамике обладать значениями разных видов, полезно иметь средство выяснения, какого же вида значениями они обладают в данный момент. Для этих целей в язык введена специальная конструкция — *отношение согласования* (*conformity relation*). В синтаксической иерархии унитарных выражений (см. § 13 гл. III) отношение согласования относится к сопоставлениям (и, следовательно, к приводимым, см. § 1 гл. IV). Отношения согласования имеют вид *boolean*, т. е. обладают только логическими значениями (однако согласно главе IV они могут в качестве приводимых занимать, например, сильные позиции вида *row of boolean* или сильные позиции операторов). Поэтому там, где в порождающих правилах фигурирует *MODE conformity relation*, возможны тупики.

Синтаксис отношения согласования следующий:  
*boolean conformity relation : soft reference to LMODE*  
*tertiary,*  
*conformity relator,*  
*RMODE tertiary.*

*conformity relator: conforms to symbol;*  
*conforms to and becomes symbol.*

*LMODE: MODE.*

*RMODE: MODE.*

Для символов *conforms to symbol* и *conforms to and becomes symbol* даются соответственно представления :: и :: =. Таким образом, отношение согласования состоит из двух частей, разделенных одним из этих двух символов. В левой его части стоит третичное (см. § 13 гл. III) выражение, занимающее мягкую позицию вида, начинающегося с *reference to* (т.е. вида имени). Позицию в правой части занимает третичное выражение произвольного вида (вспомогательные метапонятия *LMODE* и *RMODE* введены для того, чтобы в первом правиле они могли заменяться разными видами). Заметим, что в этой позиции не разрешается производить никаких приведений (отсутствует сорт позиции, т.е. случай соответствует пустому порождению *SORTETY*). Это объясняется смыслом выполнения отношения согласования, так как в процессе этого выполнения подлежит анализу сам вид выражения, стоящего в правой части.

Дадим одно определение, которое понадобится при формулировании семантики отношений согласования.

Будем говорить, что вид  $M_1$  объединен от вида  $M_2$ , если вид  $M_1$  — объединенный, и в его записи согласно синтаксису объединенных видов (см. § 1) какой-либо вид, порожденный *MODE*, есть  $M_2$  или объединен от него \*).

Опишем теперь выполнение отношения согласования, после чего дадим соответствующие пояснения. Отношение согласования выполняется следующим образом.

Шаг 1. Выполняется выражение, стоящее в его правой части, и начинает рассматриваться полученное значение.

Шаг 2. Если после начального *reference to* в виде выражения в левой части следует вид, совпадающий с

---

\*) Это определение несколько отличается от данного в [2] в силу принятой трактовки объединенных видов. Согласно [2] виды, порождаемые в данном случае *MODE*, не могут быть объединенными, поэтому последняя альтернатива отпадает.

видом рассматриваемого значения или объединенный от него, то значение отношения согласования есть *true* и шаг 4; в противном случае шаг 3.

Шаг 3. Если рассматриваемое значение — имя, то вместо него начинает рассматриваться значение, на которое оно ссылается, и шаг 2; в противном случае выполнение завершено, и значение отношения согласования есть *false*.

Шаг 4. Если символ отношения есть  $::=$ , то выполняется выражение, стоящее в левой части, и рассматриваемое значение присваивается (см. § 6 гл. III) значению (которое есть имя) этого выражения.

Рассмотрим сначала случай, когда вид выражения в правой части (т.е. порожденный *RMODE*) объединенный, а вид, порожденный *LMODE*, — не объединенный. Тогда при выполнении отношения согласования на первом шаге будет получено конкретное значение, которым обладает в данный момент объединенное выражение. Вид этого значения уже не объединенный, и если он тот самый, что и вид, порожденный *LMODE*, то можно осуществить соответствующее присваивание. Тогда значением отношения согласования является *true* (что является признаком согласованности видов, т.е. возможности присваивания), и само присваивание либо осуществляется, либо нет в зависимости от символа отношения.

Если вид рассматриваемого значения не совпадает с видом, порожденным *LMODE*, но является именем, то производится разыменование (возможно, многократное) и возврат к анализу на согласование вида полученного значения. В противном же случае вырабатывается значение *false* в качестве признака несогласованности видов, и никакого присваивания не осуществляется (даже в случае разделителя  $::=$ ).

Заметим, что разыменование производится здесь без использования общего аппарата приведенных, а предусматривается (в отличие, например, от правил вычисления значений формул) правилами определения значения отношения согласования. Подключение аппарата приведенных заблокировано тем, что позиция правой части отношения как уже отмечалось, не имеет сорта. Поэтому, если вид рассматриваемого значения есть, например, *procedure LMODE*, то никакого распроедуривания производиться не будет, и значением отношения согласования будет *false*. Это объясняется тем, что смысл исполь-

зования отношений согласования (по крайней мере в тех целях, для которых они введены в язык) сводится к работе с именами и объединенными видами (т.е. к именам, могущим ссылаться на значения разных видов).

Теперь рассмотрим противоположный случай, когда вид, порожденный *LMODE*, — объединенный, а вид, порожденный *RMODE*, — не объединенный. В этом случае, если выполнить выражение, стоящее в левой части отношения согласования, то будет получено имя, вид которого начинается с *reference to union of*. Такому имени можно присвоить значение выражения, стоящего в правой части только тогда, когда вид, порожденный *LMODE*, является объединенным от вида, порожденного *RMODE*, т.е. когда этот последний входит в состав объединения. В этом случае значением отношения согласования будет *true*, и если символ отношения есть  $:: =$ , то будет выполнено присваивание.

Пусть теперь как вид, порожденный *LMODE*, так и вид, порожденный *RMODE*, — объединенные. Такая ситуация дает наложение двух рассмотренных эффектов: можно, во-первых, получая конкретные значения «правого» выражения, «разъединять» его, и, во-вторых, проверять возможность присваиваний (и осуществлять их) в рамках «левого» объединения.

Наконец, если оба вида — не объединенные (и вид, порожденный *RMODE*, не содержит в своем составе объединенный вид после одного или нескольких *reference to*), то отношение согласования теряет интерес, так как его значение будет *true* тогда и только тогда, когда либо виды одинаковы, либо вид, порожденный *RMODE*, приводится к виду, порожденному *LMODE*, одними только разыменованиями. Это значение можно найти заранее, не выполняя отношения согласования.

Таким образом, применение отношения согласования представляет наибольший очевидный интерес в тех случаях, когда вид, порожденный *RMODE*, объединенный (или приводится к объединенному разыменованиями).

## ЛИТЕРАТУРА

1. A. van Wijngaarden (Editor), B. J. Mailoux, J. E. L. Peck and C. H. A. Koster «Draft Report on the algorithmic Language ALGOL-68», Supplement to ALGOL-Bulletin 26, Amsterdam, Mathematisch Centrum, MR 93, January 1968.
  2. A. van Wijngaarden (Editor), B. J. Mailoux, J. E. L. Peck and C. H. A. Koster «Report on the algorithmic Language ALGOL-68», Amsterdam, Mathematisch Centrum, MR101, October 1969.
  3. Д. Э. Кнут «Список неясных мест в АЛГОЛе-60», Журнал вычислительной математики и математической физики 7, 1, 1967.
  4. В. А. Васильев «О понятии правильной конструкции в алгоритмических языках», Труды 1-й Всесоюзной конференции по программированию, Киев, 1968.
  5. В. А. Васильев «О строении множества видов величин в алгоритмических языках», Журнал вычислительной математики и математической физики 10, 5, 1970.
-



Цена 43 к.

