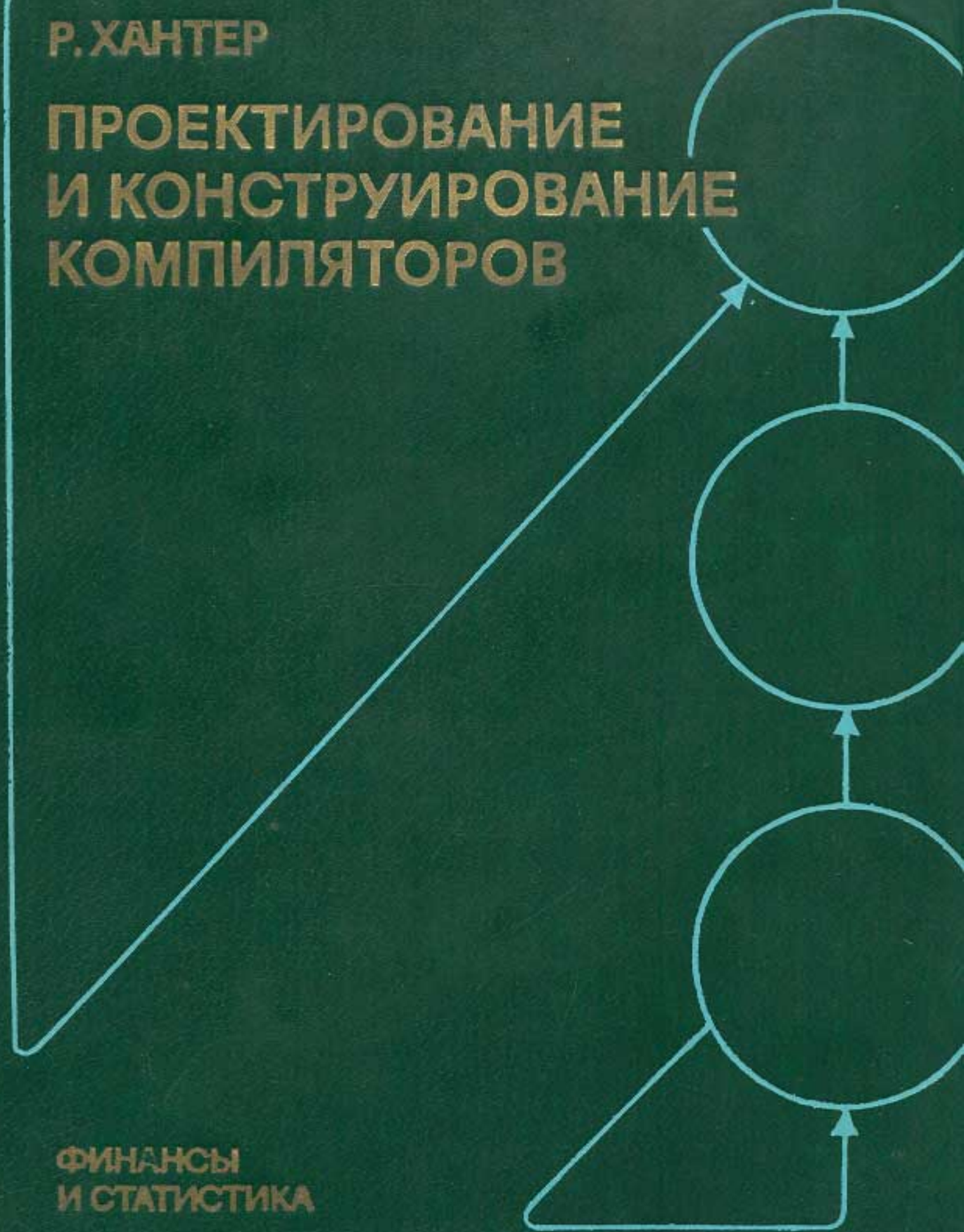
Р. ХАНТЕР

ПРОЕКТИРОВАНИЕ И КОНСТРУИРОВАНИЕ КОМПИЛЯТОРОВ



ФИНАНСЫ
И СТАТИСТИКА

ПРЕДИСЛОВИЕ К РУССКОМУ ИЗДАНИЮ

Структурные изменения парка ЭВМ, быстро возрастающая роль малых и особенно персональных ЭВМ для массовых пользователей-непрограммистов не нарушили основополагающего значения алгоритмических языков в программировании.

«Программистский бум» 80-х годов связывают прежде всего с такими алгоритмическими языками, как Паскаль, Си, Ада и др., с помощью которых развивается базовое программное обеспечение ЭВМ всех типов и разрабатываются программные средства для непрограммистов: системы запросов (простых и сложных), генераторы отчетов, графические интерпретаторы, генераторы прикладных программ, параметризованные пакеты прикладных программ, языки программирования сверхвысокого уровня.

Развитие архитектуры вычислительных машин и систем и аппаратная поддержка многих функций традиционного «софтвера» сохранили алгоритмический уровень спецификации задач. Следовательно, такие алгоритмические языки, как Ада и Паскаль, и соответствующие компиляторы остаются инструментальным средством не только профессиональных программистов, но и других специалистов, занимающихся постановкой и решением задач на ЭВМ.

Как известно, алгоритмические языки бесполезны без надежных и эффективных трансляторов. Преимущественное распространение языка Алгол 60 в нашей стране в 60—70-е годы было связано прежде всего со своевременной разработкой трансляторов для основных типов отечественных ЭВМ.

К сожалению, активная роль разработчиков компиляторов во внедрении новых перспективных алгоритмических языков заметно снизилась. Примером тому может служить язык Паскаль, опубликованный его автором еще в 1971 г. и ожидавший около 10 лет своего компилятора на ЕС ЭВМ.

Советским специалистам в области программирования предлагается перевод новой книги Р. Хантера «Проектирование и конструирование компиляторов» не только для того, чтобы в какой-то мере обеспечить подготовку высококвалифицированных разработчиков компиляторов. В большей степени эта книга будет способствовать познанию специалистами процесса компиляции, пониманию работы компиляторов и как следствие лучшему применению языков программирования. Без познания этого процесса, без необходимого теоретического уровня и практического опыта в программировании нельзя рассчитывать на своевременную разработку нового поколения языков спецификации очень высокого уровня. Поскольку развитие программирования обеспечивается применяемыми алгоритмическими языками и компиляторами с этих языков, книга Р. Хантера, детально описывающая известные методы построения компиляторов, безусловно, сыграет свою положительную роль в этом процессе.

Предлагаемая книга является первым практическим пособием, в котором все составные части компилятора и сам механизм их проектирования достаточно полно раскрыты с использованием средств, хорошо известных программистам. И хотя изложение материала опирается на еще не получивший широкого распространения Алгол 68, для чтения книги достаточно знания одного из массовых алгоритмических языков (Алгола 60, Фортрана или ПЛ/1).

Основная часть книги посвящена контекстно-свободным грамматикам, применяемым для разбора текста исходных программ. Читателю становится понятно, почему грамматики алгоритмических языков, а следовательно, и сами языки не могут быть выбраны произвольно и должны быть предельно простыми. На примере LL(1)-метода разбора автор показывает его преимущества: отсутствие возвратов, пропорциональность времени разбора длине программы, хорошие диагностические возможности, минимальный размер таблиц разбора, применимость к широкому классу языков.

Значительная роль отведена автором регулярной грамматике, или в иерархической классификации Хомского грамматике типа 3 и регулярным выражениям. Поскольку из регулярных выражений получают регулярные множества, с которыми устанавливаются алгоритмические соответствия строк языка, такие свойства регулярных множеств используются разработчиками компиляторов. Например, проверку строк языка в этом случае организуют конечным автоматом, так как существует полное соответствие между регулярными выражениями (грамматиками типа 3) и конечными автоматами. На этом основании лексический анализатор может быть представлен различными автоматами для распознавания идентификаторов, чисел, зарезервированных слов и т.п.

О многих важных свойствах компиляторов читатель получит наглядное представление благодаря хорошо подобранным и простым примерам. Специально рассмотрен случай включения в грамматику вызовов действий не только для генерирования кода, но и для построения таблиц символов и обращения к ним, размещения объектной программы в соответствии с условием и др.

Большой интерес представляет механизм распределения памяти для параллельной обработки, допускаемой в языке.

Цели создания компилятора в некоторой степени противоречивы. Так, эффективность объектного кода зависит от размера программы, трудно сделать небольшой компилятор, работающий надежно и хорошо исправляющий ошибки. Общее количество проходов, выбор промежуточного языка — эти и многие другие свойства компиляторов усложняют их разработку. Вот почему так важны четкие требования к определению исходного языка, включая уточнения грамматики, приведенные в книге Р. Хантера.

Практически ни один из вопросов, связанных с проектированием компиляторов, не упущен автором. Доходчивость и строгость изложения позволяют отнести книгу Р. Хантера к числу практических и учебных пособий, способствующих общему повышению программистской грамотности. Такая книга необходима сегодня программистам, студентам и аспирантам вузов, а также разработчикам информационных систем.

В. М. Савинков

ПРЕДИСЛОВИЕ

В книге рассматриваются в первую очередь вопросы проектирования, а затем и построения компиляторов для языков программирования высокого уровня. Главное внимание уделяется целям разработки компиляторов и средствам их достижения. Освещаются также практические аспекты построения компиляторов. В основу книги легла работа по созданию компиляторов, проводимая в последние годы на факультете вычислительных наук в Стратклайдском университете.

Книга может служить учебным пособием для студентов старших курсов, и именно такое применение она нашла в Стратклайдском университете. В ней обсуждается реализация языков с блочной структурой (Алгола 60, ПЛ/1, Алгола 68, Паскаля и Ады) с учетом особенностей конкретных языков. Описание алгоритмов приводится на реальном языке программирования (Алголе 68), но это не препятствует их пониманию, даже если студент не знаком с данным языком. Каждая глава завершается упражнениями, цель которых — помочь студентам лучше разобраться в построении компиляторов. В заключение книги читателю предлагаются схематические ответы на упражнения к гл. 1—12.

В гл. 1 рассматривается процесс компиляции в целом, а также связь между языками программирования высокого уровня и типичными ЭВМ. В гл. 2 дается формальное определение языков программирования, гл. 3 посвящается лексическому анализу, а гл. 4 и 5 — методам синтаксического анализа сверху вниз и снизу вверх.

В гл. 6 описывается применение бесконтекстной грамматики в качестве рамок для действий компилятора, а в гл. 7 — проектирование компиляторов, причем особое внимание уделяется проектированию многопроходных компиляторов.

Структура таблиц символов и видов (или типов) для языков с комплексными типами обсуждается в гл. 8. Гл. 9 знакомит читателя с локальным и глобальным распределением памяти. Гл. 10 и 11 посвящены вопросам генерирования кода и использования не зависящего от машины промежуточного кода.

В гл. 12 обсуждаются методы исправления и диагностики ошибок, а в гл. 13 высказываются некоторые соображения относительно построения надежных компиляторов.

Глава 1

ПРОЦЕСС КОМПИЛЯЦИИ

Программы, написанные на языках программирования высокого уровня (или проблемно-ориентированных языках), перед выполнением на ЭВМ должны транслироваться в эквивалентные программы, написанные в машинном коде. Языки высокого уровня появились в середине 50-х годов, примерами первых таких языков могут служить Фортран и Кобол, а примерами недавно созданных — Алгол 68, Паскаль и Ада. Программа, которая транслирует любую программу, написанную на конкретном языке высокого уровня, в эквивалентную программу на другом языке (обычно это код машины), называется *компилятором*. Компилирование программы включает *анализ* — определение предусмотренного результата действия программы и последующий *синтез* — генерирование эквивалентной программы в машинном коде. В процессе анализа компилятор должен выяснить, является ли входная программа недействительной в каком-либо смысле (т. е. принадлежит ли она к языку, для которого написан данный компилятор), и если она окажется недействительной,— выдать соответствующее сообщение программисту. Этот аспект компиляции называется *обнаружением ошибок*.

В построении компиляторов за последние двадцать пять лет достигнуты значительные успехи. Первые компиляторы использовали специальные методы, были медленными и не носили структурного характера. (Краткая история написания компиляторов изложена в [7].) В современных компиляторах применяются более системные методы; эти компиляторы относительно быстрые и строятся таким образом, чтобы как можно более четко выделять отдельные аспекты компиляции.

Существует и другой подход. Он состоит в том, чтобы вместо трансляции каждой программы в машинный код и *последующего* ее выполнения программа сначала транслировалась на промежуточный язык, а затем транслировался *и* выполнялся каждый оператор промежуточного языка (когда он встретится). Программа, транслирующая и выполняющая программу, написанную на таком языке высокого уровня, называется *интерпретатором*. Применение интерпретатора вместо компилятора имеет следующие преимущества:

1. Передавать сообщения об ошибках пользователю часто бывает легче в терминах оригинальной программы.

2. Версия программы на промежуточном языке нередко оказывается компактнее, чем машинный код, выданный компилятором.

3. Изменение части программы не требует перекомпиляции всей программы.

Диалоговые языки, такие, как Бейсик, часто реализуются посредством интерпретатора. Промежуточным языком обычно служит некая форма обратной польской записи (см. разд. 10.1). Хорошим пособием по

реализации диалоговых языков может быть работа [13]. Основной недостаток интерпретаторов заключается в том, что эти программы обычно пропускаются относительно медленно, так как операторы промежуточного кода должны транслироваться *всякий* раз, когда они выполняются, хотя временные издержки не так уж велики (они зависят от конструкции промежуточного языка). Применяется *метод смешанного кода* [15], в котором наиболее часто выполняемая часть интерпретируется. При этом экономится память, поскольку часть программы, которая должна интерпретироваться, будет, по всей вероятности, намного компактнее, чем скомпилированный код. Развитием этой идеи является «отбрасывающее» компилирование [12].

В настоящей книге речь идет в основном о компиляторах, а не об интерпретаторах, хотя многие рассматриваемые принципы применимы и к тем, и к другим. Говоря о компиляторах, мы будет называть компилируемую программу *исходной программой* (или исходным текстом), а выдаваемый машинный код — *объектным кодом* (рис. 1.1). Аналогичным образом язык высокого уровня может называться *исходным языком*, а машинный — *объектным языком*.

Рис. 1.1

В гл. 1 мы изучим характеристики языков высокого уровня и типичных ЭВМ, а также различные аспекты процесса компиляции. Для этого нам нужно иметь общее представление о построении компиляторов.

1.1. СВЯЗЬ МЕЖДУ ЯЗЫКАМИ И МАШИНАМИ

Хотя с точки зрения пользователя (и разработчика) различия между языками высокого уровня могут быть весьма значительными, мы здесь хотим подчеркнуть сходные черты этих языков, чтобы показать, какие задачи должен выполнять компилятор. Под типичными языками высокого уровня мы подразумеваем языки Бейсик, Фортран, ПЛ/1, Паскаль, Алгол 68 и (возможно, в несколько меньшей степени) Кобол. Фрагменты программ, которыми мы иллюстрируем основные положения, написаны на Алголе 68, но они в равной степени могли бы быть написаны на любом другом языке, поэтому у читателей, не знакомых с Алголом 68, не должно возникать в связи с этим никаких серьезных проблем.

Языки

Общими средствами большинства языков высокого уровня являются:

1. *Выражения и присваивания*

Значение выражения обычно можно вычислить и (помимо прочего) присвоить какой-либо переменной, например

$$A := (B + C) \times (E + F),$$

где B, C, E и F имеют соответствующие значения.

2. *Условные выражения*

Значение выражения или результат действия оператора может зависеть от логического значения (истинности), например

if $A=B$ **then** $X+Y$ **else** $X-Y$ **fi**

значением которого будет $X+Y$, если $A=B$, и $X-Y$ — в противном случае.

3. *Циклы*

Последовательность действий может выполняться несколько раз. например

```
 to 10
do
 read (i);
 print (i)
od
```

Последовательность

```
read (i);
print (i)
```

выполняется 10 раз.

4. *Ввод-вывод*

Существуют обычно простые программы для чтения или печати значений, например:

read ((x, y, z))

для чтения значений x, y и z и

print ((a, b, c))

для печати значений a, b и c.

5. *Процедуры, подпрограммы, функции*

Программе обычно можно задать определенную структуру, разделив ее на модули (и, возможно, подмодули и т. д.) и записывая каждый модуль в виде отдельной *процедуры* или *подпрограммы*. Так, если в программе время от времени производится вычисление среднего значения массива вещественных чисел, можно написать следующую процедуру:

```
proc average=([ ] real row) real:
begin real sum:=0;
for i from lwb row to upb row
do sum plusab row [i]
od;
sum/(upb row — lwb row+1)
end
```

и, допуская, что *a* объявлено как

[*1:10*] **real** *a*

можно вызвать процедуру *average*, например, следующим образом:

print (*average* (*a*))

6. *Блоки*

Всем программам нужна память для хранения значений переменных и т. п. Большинство языков дают возможность хранить переменные, появляющиеся в различных частях программы, в одной и той же памяти. В языках программирования это обычно обеспечивается с помощью *блочной структуры*, а блок определяется как часть программы, содержащая какие-либо объявления переменных, например

begin int *a, b;*
  print ((*a*×*fact*, *b*×*fact*))
end

причем объем памяти для *a* и *b* выделяется при вхождении в блок и забирается обратно при выходе из блока.

Такие языки, как Бейсик и Фортран, не имеют блочной структуры, Паскаль же ее имеет, но в ограниченной форме. Более современные языки — алголоподобные и язык министерства обороны США, Ада — имеют блочную структуру. Здесь мы будем считать в основном, что исходный язык обладает блочной структурой. Языки, не имеющие блочной структуры, легче компилировать с точки зрения распределения памяти и т. п.

На данном этапе нет смысла много говорить о различиях между языками высокого уровня, например, в отношении способов обеспечения циклов или условных выражений либо в типах обозначений операций. Различия в языках в представлении разработчика компиляторов хорошо показаны в работе [3].

Структура компилятора в значительной степени зависит от средств выражения *типа* в языке. В большинстве языков каждое возможное значение принадлежит одному из конечного или бесконечного количества типов, а каждый тип соответствует конечному или бесконечному количеству значений. Например, множество всех целых чисел, вещественных чисел или литер в каком-либо множестве литер может считаться типом. В Паскале есть подтип, соответствующий, например, целым числам в определенном диапазоне, а многие языки имеют типы-агрегаты, соответствующие массивам или записям. Виды в Алголе 68 приблизительно соответствуют типам, но являются более общими. Например, существует вид

proc (**real**) **real**

соответствующий процедуре с одним **real** параметром и одним **real** результатом. В данной книге мы употребляем оба термина: вид — в основном в контексте Алгола 68 и тип — в остальных случаях.

Считается, что некоторые языки, такие, как Алгол 68 и Паскаль, имеют *явно выраженные* или *статические типы*, т. е. типы всех значений в программе известны во время компиляции. Это привносит избы-

точность в язык и увеличивает число проверок, выполняемых в процессе компиляции. Например,

char *x:=3*

будет определена компилятором как недопустимая. В ПЛ/1, который также имеет явно выраженные типы, эта «ошибка» останется необнаруженной, так как в этом языке есть много неявных преобразований типов (например, целое в литеру).

О таких языках, как POP-2, АПЛ и ЛИСП, говорят, что они имеют неявно выраженные, или *динамические, типы,* поскольку в них типы не известны до тех пор, пока не наступит время прогона программы. Например, в POP-2 объявление

VARS X, Y;

означает, что X и Y могут принимать значения любого типа. Такое присваивание, как

$$X \rightarrow Y$$

(в POP-2 присваивания имеют направление слева направо), всегда является допустимым. Однако в

$$X + Y$$

применение обозначения бинарной операции «+» к X и Y может быть допустимым или нет в зависимости от *текущих* типов X и Y. Это можно проверить только во время прохождения программы. Такая проверка должна включаться в объектный код, что неизбежно замедляет объектную программу.

Возможно компромиссное решение: в основной части типы делают статическими, но вводят дополнительный тип (называемый, например, *any type* (любой тип) или *general* (универсальный)), который может принимать значения любых других типов языка. Это решение принято в CPL и ESPOL (подмножество Алгола 60 Барроуз).

Третий метод определения типов применяется в BCPL. Здесь имеется только один тип, который можно считать набором битов, а

$$A + B$$

означает сложение наборов битов. Программисту могут быть известны различные типы, но они неизвестны программе или реализации. Задача программиста — не допустить попыток сложения логического значения и целого числа. Во время компиляции или прогона проверка типа невозможна.

Машины

В машинном коде, как и в BCPL, концепция типа или вида отсутствует. ЭВМ имеет дело с наборами двоичных чисел независимо от их значений. Подобно языкам ЭВМ существенно отличаются друг от друга в деталях; но, чтобы показать задачу компилятора, мы можем перечислить ряд их типичных характеристик. Эти характеристики для каждой конкретной машины отчасти определяются возможностью ее использования программистом и в гораздо большей степени — стоимостными и скоростными показателями конструкции ЭВМ. Общими для большинства машин являются:

1. *Линейное запоминающее устройство*

Это — последовательность групп битов, в которых запоминается информация. Каждая группа битов называется *словом* и содержит 16, 24 или 32 бита (двоичных чисел). Место размещения слова в памяти называется его *адресом*. Адреса, как правило, идут от 0 до $2^n - 1$, где n — целое число. (В некоторых ЭВМ единицей памяти служит *байт*, который состоит из 8 бит и представляет собой поле, необходимое для записи одной литеры. Байты обычно имеют отдельные адреса).

2. *Регистры*

Как и слово основного запоминающего устройства, регистр может служить для хранения данных. Регистры используются во всех арифметических операциях, выполняемых на ЭВМ. Время, необходимое для записи значения в регистр или выборки его из регистра, намного меньше времени записи или выборки слова из основной памяти. В ЭВМ, как правило, имеется несколько регистров (8 или 16).

3. *Набор команд*

В любой машине есть набор *команд*, называемый *машинным кодом*. Каждая команда состоит из последовательности двоичных цифр и может принимать несколько параметров, которые также являются последовательностями двоичных цифр. В эти параметры обычно входит имя регистра или адрес в основной памяти. Команда (включая параметры) занимает точное число слов (или, возможно, полуслов) и может храниться в основной памяти во время выполнения программы. В одних машинных кодах команды имеют фиксированную длину (например, в серии ICL 1900), в других — переменную (например, в серии IBM 370). Используя вместо двоичных чисел мнемоническую запись кодов команд и десятичные числа вместо последовательностей двоичных цифр, можно представить типичные команды машинного кода в виде

LDX 1 4444

что значит «поместить содержимое адреса *4444* в регистр *1*». (Поскольку заранее может быть неизвестно, в какой части основной памяти будет храниться программа во время выполнения, адресные поля, как правило, бывают не абсолютными, а относительными какого-либо базового значения.) Так,

STO 2 2000

означает «записать содержимое регистра *2* по адресу *2000*»,

ADX 2 102

означает «сложить содержимое адреса *102* с содержимым регистра *2*»,

BRN 4000

«перейти к адресу *4000* и продолжать выполнение команд с этого адреса».

Остальные (перечисленные ниже) устройства ЭВМ не имеют для нас особого значения с точки зрения компиляции.

4. *Устройство управления*

Служит для интерпретации команд машинного кода и их размещения при выполнении.

5. *Арифметическое устройство*

Здесь выполняются различные арифметические операции, тесты и т. п.

Таким образом, назначение компилятора заключается в том, чтобы из исходного кода выработать выполнимый. В некоторых случаях компилятор выдает только код сборки (ассемблера) для определенной машины, а окончательная выработка машинного кода осуществляется другой программой, называемой *ассемблером*. Код ассемблера аналогичен машинному коду, но в нем для команд и адресов используется мнемоника, метки же могут применяться в качестве адресов переходов. Ассемблер выполняет довольно простую задачу, в то время как компилятор должен транслировать сложные конструкции языка высокого уровня в относительно простой машинный код или код сборки соответствующей машины.

Поэтому в работу компилятора вовлекаются два языка и, кроме них, язык, на котором написан сам компилятор. В простейших случаях это машинный код той ЭВМ, на которой он будет выполняться. Однако, как мы увидим далее, это не всегда так.

Для представления компилятора можно воспользоваться Т-образной схемой. Например, компилятор для ПЛ/1, написанный в коде ICL 1900 с целью получения кода 1900, можно изобразить так, как показано на рис. 1.2. В верхнем левом углу схемы указывается исходный язык, в верхнем правом углу — объектный язык, а внизу — язык, на котором написан компилятор. *Кросс-компилятор* выдает код для какой-либо другой машины, а не основной машины. Например, на рис. 1.3 представлен компилятор, написанный в коде ЭВМ ICL 1900, для компиляции программ на Фортране в код машины PDP-11. С помощью такого компилятора программы, написанные на Фортране, могут компилироваться на ЭВМ 1900, а пропускаться на ЭВМ PDP-11.

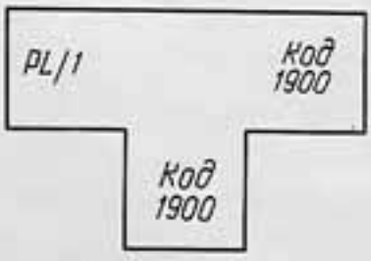


Рис. 1.2

Рис. 1.3

В настоящее время предпринимаются попытки как можно больше обособлять зависимые от машины части компилятора. Это особенно важно в тех случаях, когда компилятор предполагается перемещать из одной машины на другую. Так, если бы у нас был написанный на Фортране компилятор Паскаля, который должен выработать код, предназначенный для машины А (рис. 1.4), то, чтобы пропустить компилятор на машине А, нам прежде всего потребовалось бы транслировать его в А-код с помощью компилятора Фортрана, написанного в А-коде

Рис. 1.4

Рис. 1.5

и выдающего А-код (рис. 1.5). Из этих двух компиляторов мы смогли бы получить третий, используя второй компилятор для компиляции первого (рис. 1.6). Это можно показать, соединив вместе две Т-образные схемы в соответствии с простыми правилами (рис. 1.7), согласно которым ветви среднего звена Т должны показывать те же языки, что и корни смежных с ними соседних звеньев, а в двух верх-

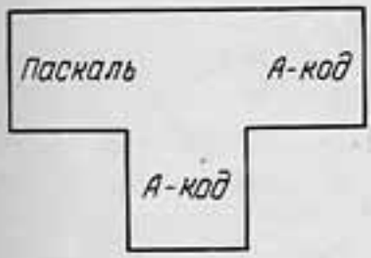


Рис. 1.6

Рис. 1.7

них звеньях в правых и левых углах должны быть записаны одинаковые языки. Можно также образовать более сложные Т-образные схемы (рис. 1.8). Идея создания Т-образной схемы принадлежит Брэтману [11].

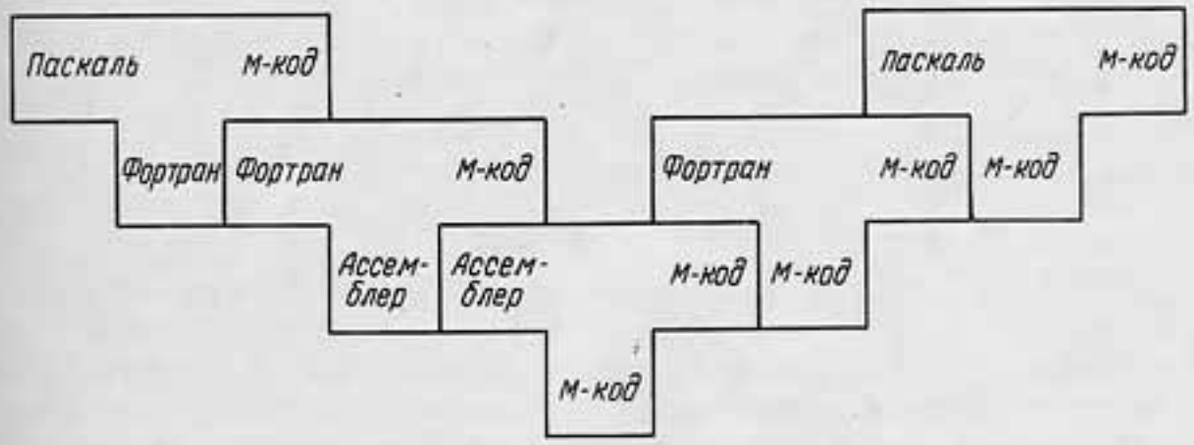


Рис. 1.8

Изменяя код, выдаваемый первоначальным компилятором в первом примере, скажем, в код для некой машины В (В-код) и допуская, что имеется компилятор Фортрана, работающий на машине В и вырабатывающий В-код, мы можем также создать для этой машины компилятор Паскаля.

1.2. НЕКОТОРЫЕ АСПЕКТЫ ПРОЦЕССА КОМПИЛЯЦИИ

Рассмотрев некоторые типичные свойства машин и языков, мы получили общее представление о тех задачах, которые выполняет компилятор. Работа компилятора складывается из двух основных этапов. Сначала он распознает структуру и значение программы, которую должен компилировать, а затем выдает эквивалентную программу в машинном коде (или коде сборки). Эти два этапа — *анализ* и *синтез*.

По идее анализ должен проводиться перед синтезом, но на практике они могут выполняться почти параллельно. Определив исходный язык, мы тем самым задаем значение его каждой допустимой конструкции (но не недопустимой). После того как анализатор распознает все конструкции в программе, он может установить, каким должен быть результат действия этой программы. Затем синтезатор вырабатывает соответствующий объектный код.

Структуру программы удобно представлять в виде дерева. Так, для программы

```
begin int x;
   x:=3;
   print (x)
end
```

на рис. 1.9 приводится схема в виде дерева, поясняющая ее структуру. Представим, что анализатор строит такое дерево (обычно его называют деревом разбора), а синтезатор просматривает его (проходя по всем вершинам в некотором заданном порядке), чтобы затем выдать машинный код.

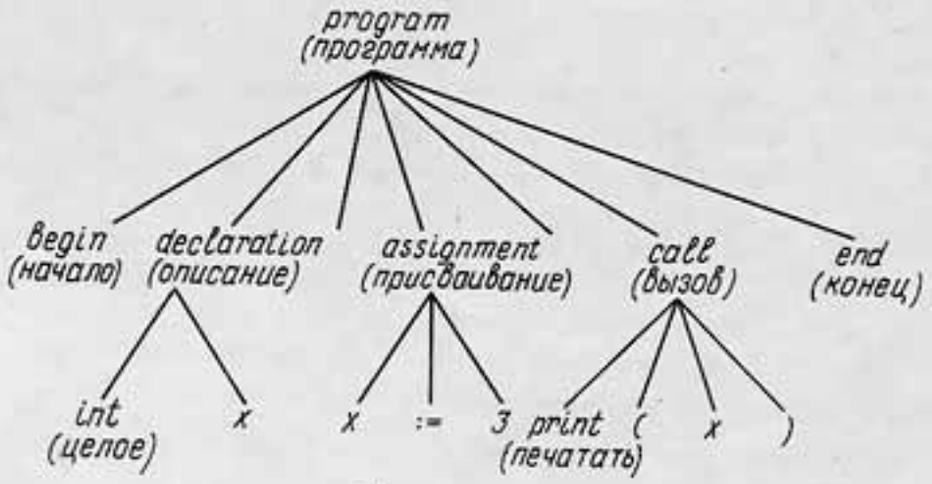


Рис. 1.9

Допустим, что конечными вершинами дерева разбора (вершинами, у которых нет ветвей) являются не отдельные литеры, а такие идентификаторы, как *print*, или такие слова языка, как **begin**. Конечную вершину с меткой *print* можно было бы заменить поддеревом, показанным на рис. 1.10, но читателю это помешает яснее представить себе структуру программы. Что касается компилятора, то в начальной фазе работы он объединяет литеры в так называемые символы, например

print, :=, **begin**, **end**, *etc.*

Эта фаза процесса компиляции называется *лексическим анализом* и рассматривается в гл. 3. Остальная часть анализа, т. е. построение дерева, называется *синтаксическим анализом*, или *разбором*.

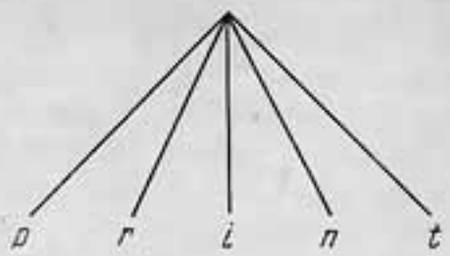


Рис. 1.10

Для проверки определенных требований языков компилятор строит таблицы. Например, в большинстве языков

$$x := y$$

допустимо только в том случае, если объявляется, что x и y имеют согласованные типы. Из такого объявления, как

int x

в таблицу, обычно называемую *таблицей символов*, вводится элемент, указывающий тип x. Вышеприведенная реализация x известна как *определяющая*. В присваивании

$$x := y$$

реализация x носит название *прикладной*, а тип x выводится из таблицы символов. Таблицы символов обычно строятся параллельно с синтаксическим анализом. Они требуются во время генерирования кода, а также для проверки действенности программ.

Могут оказаться необходимыми и некоторые другие таблицы. Во время лексического анализа идентификаторы переменной длины обычно заменяются символами фиксированной длины, а для нахождения соответствия между этими символами и первоначальными идентификаторами используется *таблица идентификаторов*. В таких языках, как Алгол 68, где комплексные виды могут определяться пользователем, подробная информация о них содержится в *таблице видов*.

В процессе компиляции, на этапе синтеза, требуется выделить память в объектной машине для записи значений, необходимых программе. Память выделяется для значений переменных, встречаемых в программе, и для рабочих операций, выполняемых, например, при вычислении выражений, подобных следующему:

$$(A+B)\times(C+D).$$

Однако во время компиляции может выделяться не весь объем памяти. Рассмотрим описание

$$[m{:}n]\,\textbf{int}\ A$$

Допустив, что значения m и n неизвестны при компиляции, нельзя выделить соответствующий объем памяти для элементов A до начала прогона. Поэтому компилятор, вместо того чтобы выделить память для элементов A, выдаст код для выделения соответствующего объема памяти в ходе прогона. Объем памяти, выделяемый во время компиляции, обычно называется *статическим*, а объем памяти, выделяемый во время прогона, — *динамическим*.

Запоминающее устройство, как правило, организуется по принципу магазина (первым считывается последнее записанное слово) и носит название *стека* (см. гл. 9).

Генерирование кода описывается в гл. 10 и 11. В гл. 10 мы рассмотрим получение не зависящего от машины промежуточного кода, а в гл. 11 — способы его отображения на реальной машине. Введение промежуточного языка связано с попыткой разделить зависимые и не зависимые от машины аспекты генерирования кода. Промежуточный код может состоять из операторов вида

ADD address 1, address 2, address 3

что значит «сложить величины, записанные по адресам *1* и *2*, и поместить результат по адресу *3*». Промежуточный язык не является, по всей вероятности, не зависимым от исходного языка (хотя были предприняты попытки создания универсальных промежуточных языков, см. [48]). Он отражает основные действия исходного языка и имеет достаточно высокий уровень, чтобы его можно было эффективно реализовать на типичной ЭВМ.

Во многих компиляторах нет явного разделения этих двух аспектов генерирования кода. Но если компилятор предполагается сделать переносимым, то желательно как можно полнее разделить его зависимые и не зависимые от машины части.

Конечно, выданный окончательный код должен иметь то же значение, что и программа. Однако выражения разобьются на основные компоненты, виды исчезнут, а все значения окажутся представленными как наборы двоичных знаков. Структура программы (условные выражения и циклы) будет представлена тестами, переходами и метками. Программа станет гораздо менее понятной, но появится возможность выполнять ее на какой-нибудь конкретной машине.

В этом разделе мы попытались объяснить некоторые аспекты процесса компиляции. Подробнее организация различных фаз компилятора и их соответствие друг другу описываются в следующих главах.

1.3. ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА

Как уже упоминалось выше, компилятор в некотором смысле определяется тем языком высокого уровня, который он принимает, и машинным кодом (в более общем смысле — другим языком), который он создает. Из этого, однако, не следует, что, например, все компиляторы Фортрана для машин серии PDP-11 должны быть одинаковыми. Конечно, вряд ли можно ожидать, что два компилятора, написанные разными разработчиками (или группами разработчиков), окажутся одинаковыми, хотя они и должны выдавать идентичный, или эквивалентный, объектный код из одного и того же исходного кода. Два компилятора, реализующие один и тот же язык на одной и той же машине, могут отличаться хотя бы потому, что разработчики преследовали различные цели при их построении. Этими целями могут быть:

1) получение эффективного объектного кода;
2) разработка небольших объектных программ;
3) минимизация времени компиляции программ;
4) разработка компилятора как можно меньшего размера;
5) создание компилятора, обладающего широкими возможностями обнаружения и исправления ошибок;
6) обеспечение надежности компилятора.

К сожалению, эти цели в какой-то степени противоречивы. Почти наверняка компилятор, который должен выдавать эффективный код,

будет немного (и даже намного) медленней. Для осуществления некоторых стандартных методов оптимизации [20], таких, как устранение избыточности кода, исключение последовательностей кода из циклов и т. д., может потребоваться очень много времени. Компилятор, в котором выполняются сложные программы оптимизации, является также бо́льшим по объему. Размер компилятора может влиять и на время, необходимое ему для компиляции программы. В процессе работы часто невозможно оптимизировать время и объем памяти одновременно. Обычно одно осуществляется за счет другого. Разработчик компилятора должен заранее решить, что именно он предпочитает оптимизировать, а затем уже приступить к оптимизации, полностью отдавая себе отчет в том, к чему это может привести. Кроме того, у него может возникнуть желание потратить как можно меньше времени на написание компилятора, что само по себе является препятствием к включению некоторых сложных методов оптимизации.

Программисты ожидают от компилятора вспомогательного сообщения об ошибке, когда они представляют недопустимую программу. Их не удовлетворит формальный ответ, что программа «написана не на том языке, для трансляции которого предназначен компилятор». Они также предполагают, что компилятор, обнаружив ошибку, не перестанет производить компиляцию программы, а будет продолжать анализировать ее в поисках дальнейших ошибок. Компиляторы крайне отличаются по своим возможностям обнаружения и исправления ошибок, хотя это их свойство не противоречит перечисленным выше целям проектирования. Конечно, компилятор, выдающий полезные сообщения и элегантно выходящий из ситуации ошибки, имеет как следствие несколько больший размер, но это вполне компенсируется повышением эффективности работы программиста. В результате для компиляции правильных программ такому компилятору не потребуется больше времени. Однако проблема исправления ошибок (особенно синтаксических) не проста и полностью пока еще не решена. Более подробно эта проблема рассматривается в гл. 12.

Несомненно, обеспечение надежности должно быть первостепенной задачей при создании любого компилятора. Компиляторы часто представляют собой очень большие программы, а современное состояние этой области науки и техники не позволяет дать формальное подтверждение правильности компилятора. Наибольшее значение здесь имеет хороший общий проект. Если различные фазы процесса сделать относительно различимыми и каждую фазу построить как можно проще, то вероятность создания надежного компилятора должна возрасти. Надежность компилятора повышается и в том случае, если он базируется как можно полнее на ясном и однозначном формальном определении языка и если используются такие автоматические средства, как *генераторы синтаксических анализаторов*. Конечно, компилятор должен быть всесторонне проверен, причем не только в целом, но и его отдельные части до их сборки. Вопросы надежности обсуждаются также в гл. 13.

Нередко фирмы-изготовители предлагают для какого-либо языка, применимого в серии машин, не один компилятор, а несколько. Так, компанией «Инглиш электрик» были предложены для ЭВМ KDF9 компиляторы Whetstone и Kidsgrove. Компилятор Whetstone является быстрым в компиляции и медленным в прогоне, а Kidsgrove — медленным в компиляции, но выдает очень эффективный машинный код. Идея за-

ключалась в том, чтобы пользователи разрабатывали свои программы с помощью компилятора Whetstone, а затем для работы перекомпилировали их с помощью компилятора Kidsgrove. В ИБМ также создали несколько компиляторов ПЛ/1 для машин серии 370.

Цели проектирования компилятора часто зависят от той среды, в которой он должен использоваться. Если он предназначается в основном для студентов, которые редко пропускают свои программы после удовлетворительного завершения фазы их разработки, эффективность машинного кода будет иметь меньшее значение, чем скорость компиляции и возможности обнаружения ошибок. В производственной сфере степень важности этих свойств может быть обратной. В учебной среде пакетный компилятор, который остается в основной памяти, пока ряд программ компилируется и прогоняется, позволяет эффективно применять имеющуюся аппаратуру. В построении компилятора большую роль играет решение о числе *проходов*, которое ему придется выполнять. Каждое прочитывание исходного текста или его версии компилятором считается проходом. *Однопроходные компиляторы* привлекательны из-за их простоты. Однако не все языки позволяют выполнять однопроходную компиляцию. Со структурной точки зрения отдельные фазы компиляции могут быть выполнены аккуратнее и проще в разные проходы. Чтобы обеспечить возможность однопроходной компиляции, некоторые разработчики реализуют только какое-либо подмножество языка, настаивая, например, на том, чтобы идентификаторы описывались до их употребления в исходных программах.

Создание *многопроходного компилятора* обычно связано с проектированием промежуточных языков для версий исходного текста, существующих между проходами. Строятся также таблицы какого-либо прохода, которые могут понадобиться для просмотра в последующем проходе или проходах. Поэтому организация многопроходного компилятора выглядит усложненной по сравнению с организацией однопроходного компилятора, хотя отдельные проходы могут быть относительно простыми. Они позволяют разделить задание между рядом исполнителей или групп и не требуют сложных взаимосвязей между ними. Наконец, следует отметить, что не все компиляторы имеют фиксированное число проходов. В некоторых из них число проходов, необходимых для выполнения программы, зависит от того, все ли идентификаторы были описаны до их употребления и т. д., или от степени оптимизации, затребованной программистом. Мы продолжим обсуждение этой темы в гл. 7.

Создатель компилятора должен выбрать язык, на котором он будет писать свой компилятор. Все в большей степени для этого применяются языки высокого уровня, универсальные языки, такие как Алгол 68, Паскаль или ПЛ/1, либо так называемые языки программного обеспечения, например BCPL или ПЛ/360. Следует упомянуть и об идее создания компилятора для какого-либо языка на самом языке. В этом случае разработчик должен мыслить только в терминах одного языка, а компилятор можно использовать для самокомпиляции, что само по себе является хорошей проверкой. Кроме того, это — путь для создания переносимого компилятора.

Упражнения

1.1. Приведите какие-нибудь другие типичные конструкции языков высокого уровня и укажите, какого типа команды в машинном коде будут генерироваться при их трансляции.

1.2. Компилятор транслирует Паскаль в машинный код ICL 1900, а написан на Алголе 68. Какое еще программное обеспечение потребуется для того, чтобы пропускать программы Паскаля на ЭВМ ICL 1900?

1.3. Некоторые компиляторы генерируют код для проверки индексов массивов во время прогона, чтобы убедиться в том, что они находятся в объявленных границах массива. Рассмотрите аргументы за и против реализации таких проверок:
а) в процессе разработки программы;
б) в процессе рабочих прогонов.

1.4. Требуется разработать компилятор Фортрана за относительно короткое время. Как вы думаете, какие из шести целей проектирования, перечисленных в разд. 1.3, будет труднее всего осуществить?

1.5. Какие из следующих средств языка высокого уровня, по вашему мнению, окажутся полезными для разработчика компилятора:
а) массивы?
б) арифметические действия с вещественными числами?
в) процедуры?

1.6. Приведите доводы в пользу применения в учебной среде интерпретатора, а не компилятора.

1.7. Какие вероятные проблемы вы видите в методе KDF 9, предлагающем разные компиляторы для этапов трансляции и прогона рабочей программы?

1.8. Некоторые языки позволяют программисту присваивать относительный приоритет обозначениям операций. Как может это сказаться на дереве разбора конкретной программы?

1.9. Каковы, по вашему мнению, основные цели проектирования компилятора для языка, на котором реализуется программное обеспечение?

1.10. Какое средство вы ожидаете найти в компиляторе, предназначенном для такого диалогового языка, как Бейсик?

Глава 2

ОПРЕДЕЛЕНИЕ ЯЗЫКА

Прежде чем приступить к созданию компилятора, необходимо (или по меньшей мере крайне желательно) иметь четкое и однозначное определение исходного языка. К сожалению, хотя и существуют хорошо разработанные методы спецификации некоторых аспектов языков программирования, полные и ясные определения реализуемых языков часто отсутствуют. Более того, формальное определение языка не обязательно используется создателем компилятора. В настоящей главе мы знакомим читателя с несколькими методами определения языков программирования. Подробнее эта тема освещена в [44]. Мы также рассматриваем здесь проблему разбора — как выяснить, принадлежит ли какая-либо последовательность символов конкретному языку или нет.

2.1. СИНТАКСИС И СЕМАНТИКА

Можно представить себе язык состоящим из ряда строк (последовательностей символов). В описании языка определяется, какие строки принадлежат этому языку (синтаксис языка), и значение этих строк (семантика языка). Синтаксис для конечного языка (т. е. состоящего только из конечного числа строк) можно специфицировать, задав список строк. Например, язык может содержать строки

abc
xyz

Строки, принадлежащие языку, обычно называются предложениями. Большинство представляющих для нас интерес языков включает бесконечное число предложений, так что их синтаксис нельзя определить путем перечисления этих предложений. К примеру, предложениями языка могут быть «все строки, состоящие только из нулей и единиц». Так,

101 1100 11001

может принадлежать языку, и в этом случае цитируемая фраза представляется достаточным определением языка. Однако русский[1] язык не очень годится для спецификации синтаксиса более сложных языков, поэтому, как правило, применяется более формальный метод определения синтаксиса языков программирования, о чем мы узнаем позже.

Семантика задает значение всем предложениям языка. Например,

begin int *k; read (k); print (k+1)* **end**

является предложением Алгола 68, и исходя из семантики этого языка можно вывести значение данной программы в терминах описания иден-

[1] В оригинале — английский.—*Примеч. пер.*

тификатора, считывания значения для него, вычисления и печати выражения. Семантика языка, как и синтаксис, обычно описывается довольно формально. Однако методы определения семантики не так хорошо разработаны, как методы определения синтаксиса, и нужны дальнейшие исследования в этой области, чтобы получить полностью удовлетворительное формальное определение языков программирования. Развитие указанного направления может позволить создавать компиляторы автоматически на основе формального определения семантики и синтаксиса языка. В следующем разделе мы рассмотрим задачу спецификации синтаксиса языка.

2.2. ГРАММАТИКИ

Начнем с терминологии. Фигурные скобки используются для обозначения множеств, например

$$\{1, 2, 3\}$$

обозначает множество, содержащее целые числа 1, 2 и 3. Объединение ($\cup$) и пересечение ($\cap$) множеств определяются как обычно:

$$\{1, 2, 3\} \cup \{3, 4, 5\} = \{1, 2, 3, 4, 5\},$$
$$\{1, 2, 3\} \cap \{3, 4, 5\} = \{3\}.$$

Мы говорим, что множество A включает ($\supseteq$) множество B, если каждый элемент B является элементом A, например

$$\{3, 4, 5\} \supseteq \{3\}.$$

Аналогично множество B включается ($\subseteq$) в множество A, если каждый элемент B имеется также и в A:

$$\{3\} \subseteq \{3, 4, 5\}.$$

Мы употребляем символ $\in$, чтобы показать, что элемент содержится в множестве, например

$$3 \in \{3, 4, 5\}.$$

Пустое множество обозначается посредством $\varnothing$. Поэтому

$$\{1, 2\} \cap \{3, 4, 5\} = \varnothing.$$

Если A есть множество

$$\{2, 4, 9, 8\}$$

и мы определяем B как

$$B = \{x \mid x \in A \text{ и } x \text{ есть четное число}\},$$

то B будет множеством

$$\{2, 4, 8\}.$$

B задается с помощью предиката, в данном случае это «$x \in A$ и x есть четное число».

Мы определяем *алфавит* (или *словарный состав*) как множество символов. Например, им может быть латинский или греческий алфавит либо десятичные цифры 0 — 9. Представим алфавит

$$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

Если A есть алфавит, A^* (замыкание A) используется для обозначения множества всех строк (включая пустую строку, состоящую из нулей), составленных из символов, входящих в A. A^+ обозначает множество всех строк (исключая пустую строку), состоящих из символов, входящих в A. Пустая строка обычно обозначается с помощью ε.

Синтаксис языка можно специфицировать, пользуясь системой изображения множеств, например

$$L=\{0^n1^n \mid n\geqslant 0\}.$$

Иными словами, этот язык включает строки, состоящие из нуля или нулей и того же числа последующих единиц. Пустая строка включается в язык.

Таким же образом можно специфицировать следующие синтаксисы:

1. $\{a^nb^nc^n \mid n\geqslant 0\}$;
2. $\{a^mb^n \mid m,\ n\geqslant 0\}$,

т. е. число a с последующим числом (не обязательно тем же самым) b, например

$$aaabb,$$

будет принадлежать этому языку;

3. $\{x^m \mid m$ — простое число$\}$;
4. $\{a^mb^nc^p \mid m=n$ или $n=p\}$.

Все эти синтаксисы намного проще синтаксисов большинства языков программирования; синтаксис же языка программирования лучше специфицировать с помощью *грамматики*. Грамматика состоит (частично) из набора правил для получения предложений языка. Возьмем, например, синтаксис, определенный ранее в этом разделе,

$$\{0^n1^n \mid n\geqslant 0\}$$

и воспользуемся следующими правилами:

1. $S\to 0S1$,
2. $S\to\varepsilon$.

Чтобы вывести предложение этого языка, поступим следующим образом. Начнем с символа S и заменим его на $0S1$ или ε. Если S опять появится в полученной строке, его опять можно заменить с помощью одного из этих правил, и т. д. Полученная таким образом любая строка, не содержащая S, является предложением этого языка. Например,

$$S\Rightarrow 0S1\Rightarrow 00S11\Rightarrow 000S111\Rightarrow 000111.$$

Последовательность указанных шагов называется *выводом* строки 000111, а символ $\Rightarrow$ служит для разделения шагов вывода. Все предложения данного языка можно вывести, руководствуясь приведенными выше двумя правилами, и любая строка, которую нельзя вывести с их помощью, не является предложением этого языка. Грамматику часто (и справедливо) называют системой перезаписи. Определим теперь грамматику более формально.

Грамматика определяется как четверка

$$(V_T,\ V_N,\ P,\ S),$$

где V_T — алфавит, символы которого называются *терминальными* (или просто *терминалами*); V_N — алфавит, символы которого называются

нетерминальными (или *нетерминалами*), и у V_T, и V_N нет общих символов, т. е.

$$V_T \cap V_N = \varnothing.$$

V определяется как $V_T \cup V_N$.

P есть набор *порождающих правил*, каждый элемент которых состоит из пары (α, β), где α находится в V^+, а β в V^*. α обычно известна как левая часть правила, а β — как правая, и правило записывается следующим образом:

$$\alpha \to \beta.$$

$S \in V_N$ и называется *начальным символом* (или *аксиомой*). Этот символ является отправной точкой в получении любого предложения языка.

Грамматикой, генерирующей язык

$$\{0^n 1^n \mid n \geqslant 0\}$$

является G_0, где

$$G_0 = (\{0, 1\}, \{S\}, P, S)$$

и

$$P = \{S \to 0S1, S \to \varepsilon\}.$$

Грамматика, генерирующая строки в наборе

$$\{a^m b^n \mid m, n \geqslant 0\},$$

имеет вид

$$G_1 = (\{a, b\}, \{S, A, B\}, P, S).$$

Здесь элементами R будут

$$\begin{aligned} S &\to AB \\ A &\to aA \\ A &\to \varepsilon \\ B &\to bB \\ B &\to \varepsilon \end{aligned}$$

Начав с символа S и применяя последовательно по одному из правил замены нетерминала в выводимой строке, мы можем генерировать строку *aaabb*:

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaaAB \Rightarrow aaaB \Rightarrow aaabB \Rightarrow aaabbB \Rightarrow aaabb.$$

Каждая строка, которую можно вывести из начального символа (например, *aAB*, *aaaB*), называется сентенциальной формой. Предложение есть сентенциальная форма, содержащая только терминальные символы.

$$\gamma \overset{+}{\underset{G}{\Rightarrow}} \delta$$

означает, что строку (символов) δ можно вывести из строки γ, однократно или неоднократно применяя правила грамматики G. (G можно опустить, если ясно, какая грамматика предполагается). Из изложенного видно, что

$$aAB \overset{+}{\underset{G_1}{\Rightarrow}} aaabB.$$

Аналогичным образом

$$\gamma \overset{*}{\underset{G}{\Rightarrow}} \delta$$

означает, что строку δ можно вывести из строки γ с помощью нуля или более приложений правил грамматики G. (G также можно опустить при соответствующих обстоятельствах.) Например,

$$aAB \overset{*}{\underset{G_1}{\Rightarrow}} aaabB,$$

$$aAB \overset{*}{\underset{G_1}{\Rightarrow}} aAB.$$

Там, где правило используется *точно* один раз, применяется символ $\Rightarrow$, например

$$AB \underset{G_1}{\Rightarrow} aAB$$

или, если опускается G_1,

$$AB \Rightarrow aAB.$$

В настоящей книге для обозначения терминалов грамматики мы будем употреблять прописные буквы (или строки прописных букв), а для обозначения нетерминалов — заглавные буквы (или строки заглавных букв). Там, где может возникнуть неоднозначность в последовательностях букв, обозначающих терминалы или нетерминалы, для разделения символов в правиле мы введем пробелы. Строки терминалов и/или нетерминалов представим с помощью греческих букв.

Как правило, предложения языка можно генерировать посредством более чем одной грамматики; две грамматики, генерирующие один и тот же язык, называют эквивалентными. Например, G_1 эквивалентна G_2, определенной следующим образом:

$$G_2 = (\{a, b\}, \{S, Y\}, P, S),$$

где элементами P являются

$$\begin{array}{ll} S \to aS & Y \to b \\ S \to a & Y \to bY \\ S \to b & S \to \varepsilon \\ S \to bY & \end{array}$$

С точки зрения составителя компилятора одна грамматика может быть гораздо удобнее другой в качестве основы для фазы синтаксического анализа компиляции.

В приводимых выше примерах левые части правил всегда состояли из одного нетерминала. Однако из определения грамматики вытекает, что необязательно должно быть так. Возьмем грамматику

$$G_3 = (\{a\}, \{S, N, Q, R\}, P, S)$$

со следующими элементами P:

$$\begin{array}{ll} S \to QNQ & RQ \to NNQ \\ QN \to QR & N \to a \\ RN \to NNR & Q \to \varepsilon \end{array}$$

Генерироваться будет

$$\{a^{(2^n)} \mid n \geqslant 0\}.$$

Типичный вывод:

$$\begin{aligned} &S \Rightarrow QNQ \Rightarrow QRQ \Rightarrow QNNQ \\ &\Rightarrow QRNQ \Rightarrow QNNRQ \Rightarrow QNNNNQ \\ &\overset{+}{\Rightarrow} aaaa. \end{aligned}$$

Рассмотрев, какие роли играют N, Q и R в вышеприведенном выводе, читатель сможет убедиться в том, что генерированный язык и

есть объявленный язык. Следует заметить также, что R служит для дублирования N.

Ограничим типы правил, которые могут появляться в грамматике. Это позволит нам определить ряд специальных классов грамматик. Одна из стандартных классификаций известна как *иерархия Хомского*. Ее можно описать следующим образом.

Любая грамматика определенного ранее вида называется грамматикой типа 0.

Если, однако, грамматика обладает тем свойством, что для всех правил вида

$$\begin{gathered}\alpha \to \beta, \\ |\alpha| \leqslant |\beta|.\end{gathered}$$

где $|\alpha|$ обозначает длину α, т. е. число символов в α, и то же для $|\beta|$, то она называется грамматикой *типа 1*, или *контекстно-зависимой*.

Если грамматика имеет такое свойство, что все левые части состоят из одного нетерминального символа, то ее называют грамматикой *типа 2*, или *контекстно-свободной*.

В том случае, когда каждое правило грамматики имеет одну из форм

$$\begin{gathered}A \to a \\ A \to aB\end{gathered}$$

где a — терминальный символ, а A и B — нетерминальные, эту грамматику называют *грамматикой типа 3, выровненной вправо*, или *регулярной*.

Выровненная влево грамматика определяется аналогично предыдущей, т. е. все ее правила имеют одну из форм

$$\begin{gathered}A \to a \\ A \to Ba\end{gathered}$$

Она также называется грамматикой типа 3, или регулярной.

Очевидно, что эта иерархия — включающая, т. е. все грамматики типа 3 являются грамматиками типа 2, все грамматики типа 1 являются грамматиками типа 0 и т. д. Иерархии грамматик соответствует иерархия языков. Например, если язык можно генерировать с помощью грамматики типа 2, то его называют языком типа 2, а если с помощью грамматики типа 3,— то его называют языком типа 3. Эта иерархия — опять включающая. Включения также справедливы в том, что существуют языки, которые относятся к типу i, но не к типу $(i+1)$ $(0 \leqslant i \leqslant 2)$.

Можно показать, например, что язык, генерированный посредством G_3, — контекстно-зависимый, а не контекстно-свободный, т. е. не существует контекстно-свободной грамматики, которая генерирует этот язык. Аналогично язык, генерированный посредством G_0, — контекстно-свободный и нерегулярный. Однако тот факт, что язык можно генерировать с помощью нерегулярной контекстно-свободной грамматики, необязательно означает, что он нерегулярный. Например, грамматика G_1 — контекстно-свободная, но нерегулярная, а язык, генерированный посредством ее, — регулярный, так как его также можно генерировать посредством грамматики G_2. (Строго говоря, регулярной будет $L(G_2) - \varepsilon$, но принято расширять определения грамматики типа 3 и включать правила вида $S \to \varepsilon$, где S — начальный символ.) Иерархия Хомского важна с точки зрения языков программирования. Чем меньше ограничений в грамматике, тем сложнее ограничения, которые можно наложить на генерируемый язык.

Грамматики типа 3 можно использовать для описания некоторых свойств языков программирования. Например, для генерирования идентификаторов по определению многих языков программирования можно воспользоваться следующими правилами:

IDENT→letter
IDENT→letter REST
REST →letter
REST →digit
REST →letter REST
REST →digit REST

где буква и цифра являются терминалами. Иногда удобно объединять правые части правил, имеющих одинаковые левые части. Вышеприведенную грамматику можно было бы также записать в виде

IDENT→letter|letter REST
REST →letter|digit|letter REST|digit REST

Здесь вертикальную черту следует читать как «или».

Многие «локальные» средства языков программирования, например константы, слова языка и строки, представляются с помощью грамматик типа 3. Однако можно показать, что грамматики типа 3 генерируют строго ограниченные типы языков. Определим теперь *регулярные выражения* и скажем, не доказывая, что грамматики типа 3 генерируют все регулярные выражения без исключения (см. [2], с. 118).

В алфавите A к регулярным выражениям относятся следующие:

1. Элемент A (или пустая строка).

Если P и Q — регулярные выражения, то ими будут также и выражения

2. PQ (Q следует за P).
3. $P|Q$ (P или Q).
4. P^* (нуль или более экземпляров P).

В алфавите $\{a, b, c\}$

$$a^*b|ca^*$$

— регулярное выражение, которое описывает язык, включающий следующие строки (помимо прочих):

aab
c
caa
ab
ca

Если предположить, что регулярные выражения построены с помощью трех знаков операций: конкатенации (представленной соединением), | и *, то при написании регулярных выражений знак * будет обладать главным приоритетом, за ним последует конкатенация, а затем знак |.

Операция | (иногда представляемая как +) является коммутативной и ассоциативной, т. е. для регулярных выражений P, Q, R:

$P|Q=Q|P$ (коммутативная),
$(P|Q)R=P|(Q|R)$ (ассоциативная).

Конкатенация является ассоциативной, но не коммутативной.

Для изменения приоритетов, обычно ассоциируемых с этими обоз-

начениями операций, можно воспользоваться скобками. Так, в алфавите $\{a, b\}$

$$(aab|ab)^*$$

— регулярное выражение, которое описывает язык, включающий строки:

ε
aababaab
ababab
aabaabaabab

Регулярное выражение, описывающее идентификатор, имеет вид

$$L(L|D)^*,$$

где L обозначает букву, а D — цифру.

Регулярное выражение генерирует регулярное множество. Например, регулярное выражение $(a|b)c$ генерирует регулярное множество $\{ac, bc\}$. Ясно, что регулярное множество можно генерировать с помощью не только одного регулярного выражения (хотя бы потому, что «|» является коммутативной). Мы обычно не делаем различия между регулярным выражением и генерируемым им множеством.

Другие свойства регулярных множеств:

1. Существует алгоритм, определяющий принадлежность строки заданному регулярному множеству (специфицированному регулярным выражением).

2. Существует алгоритм, который определяет, не генерируют ли два регулярных выражения одно и то же регулярное множество.

Эти свойства весьма полезны с точки зрения разработчика компилятора.

Тем не менее у регулярных выражений есть свои ограничения. Например, регулярное выражение не может задавать шаблоны скобок произвольной длины, и, следовательно, их нельзя генерировать с помощью грамматики типа 3. Рассмотрим язык, состоящий из строк открывающих и закрывающих скобок (плюс пустая строка), обладающих следующими свойствами:

1. При чтении слева направо число встреченных закрывающих скобок никогда не превышает число встреченных открывающих скобок.

2. В каждой строке содержится одинаковое число открывающих и закрывающих скобок.

Например, следующие строки будут находиться в языке:

((( )))
() () (())
(() () (() ()))

а приводимые ниже — нет:

()) (() — правило 1,
((() ()) — правило 2.

Не существует способа выражения этого языка в виде регулярного выражения (см. [45], с. 72) или его генерирования с помощью грамматики типа 3.

Однако этот язык генерируется контекстно-свободной грамматикой, обладающей следующими правилами:

$$S \rightarrow (S)$$
$$S \rightarrow SS$$
$$S \rightarrow \varepsilon$$

В большинстве языков программирования имеются пары скобок, которые необходимо согласовывать, например

( ), [ ], begin end, if fi, do od

и, конечно, каждой открывающей скобке должна соответствовать закрывающая скобка. Так,

begin( )end

представляет собой правильную структуру скобок, а

begin (end)

— неправильную. Контекстно-свободная грамматика может специфицировать ограничения подобного вида. Контекстно-свободные грамматики продвинулись далеко в описании синтаксических свойств языков программирования, и их обычно используют при компиляции как основу для фазы синтаксического анализа. И все же у типичных языков программирования есть некоторые свойства, которые нельзя выразить посредством контекстно-свободной грамматики. Например, присваивание

$$X := Y$$

может быть допустимым, если объявлено, что X и Y имеют соответствующие типы. Если X и Y были описаны как

int X;**char** Y

присваивание (по крайней мере, в Алголе 68) недопустимо. Условие такого вида не может быть специфицировано контекстно-свободной грамматикой, и компиляторы обычно выполняют проверку типа не на фазе формального синтаксического анализа. Однако в следующем разделе мы покажем, что идею контекстно-свободной грамматики можно расширить, включив некоторые контекстно-зависимые свойства языков.

Таким образом, оказывается, что чем более универсален класс используемой грамматики, тем больше средств типичных языков программирования мы можем описать. Однако чем универсальнее грамматика, тем сложнее должна быть машина (или программа), которая применяется для распознавания строк соответствующего языка. В гл. 3 мы увидим, что с грамматикой типа 3 ассоциируется класс распознавателей, известный как конечный автомат или машина с конечным числом состояний, между которыми происходит передача управления по мере считывания символов строки, причем строка принимается или нет в зависимости от того, какого состояния машина достигает в итоге. Для языка, генерируемого с помощью контекстно-свободной грамматики, необходим (вообще) автомат магазинного типа, т. е. конечный автомат плюс стек, а для контекстно-зависимых языков — линейный автомат с ограничениями, т. е. машина Тьюринга с конечным объемом ленты. Наконец, языку типа 0 требуется машина Тьюринга в качестве распознавателя.

Полное описание этих машин, а также доказательства эквивалентности различных классов языков и соответствующих им автоматов приведены в [27].

2.3. ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Под формальным определением языка программирования мы понимаем полное описание его синтаксиса и семантики.

Формальные определения языков программирования находят применение при проверке правильности программ и в проектировании языков. Правда, существуют мнения, что некоторые языки содержат средства, которые отражают метод формального определения, использованный на этапе проектирования. Однако с нашей точки зрения можно привести две основные причины, по которым желательно иметь формальные определения языков программирования.

1. Программисты хотят иметь возможность получать авторитетные ответы на те вопросы, которые у них могут возникать в отношении синтаксиса и семантики языка. Сведения о языке содержатся в учебниках и руководствах, но они часто неоднозначны или не освещают всех тонкостей.

2. Разработчикам компиляторов требуется точное определение языка, который они реализуют, и предпочтительно в легко реализуемой форме.

Эти две причины представляются противоречивыми. Первая подразумевает, что определение должно быть удобочитаемым и что программисту, знакомому с ним, будет нетрудно получить ответ относительно языка. Согласно второй причине определение должно иметь такой вид, чтобы на его основе можно было легко и при наименьшем вмешательстве человека построить (по крайней мере) анализатор. В гл. 4 мы увидим, что наиболее естественные (и удобочитаемые) грамматики не всегда позволяют автоматически построить анализатор. В частности, некоторые методы разбора, как правило, требуют преобразования грамматики перед ее использованием для построения синтаксического анализатора.

Сообщение об Алголе 60 явилось одной из первых попыток формального определения языка. Большая часть синтаксиса была описана с помощью контекстно-свободной грамматики, а остальная его часть и семантика — на английском языке. В первом сообщении содержалось много неоднозначностей, и даже пересмотренное сообщение [46] оказалось несвободным от них [33].

Дальнейшая разработка методов формального определения представлена сообщением об Алголе W [8], в котором сделана попытка включить в формальную часть синтаксиса некоторую информацию о типе. В пересмотренном сообщении об Алголе 68 [55] для определения всего синтаксиса языка использовалась двухуровневая грамматика (W-грамматика, названная в честь ее изобретателя грамматикой А. ван Вейнгаардена). Идея применения двухуровневой грамматики заключается в том, что, как и правила обычной грамматики, обеспечивающие конечный способ описания языка, который состоит из бесконечного числа строк, в сообщении об Алголе 68 (здесь и далее под Алголом 68 и сообщением об Алголе 68 мы подразумеваем пересмотренный Алгол 68 и пересмотренное сообщение об Алголе 68 соответственно) вторая грамматика применяется для генерирования бесконечного чис-

ла правил, которые в свою очередь генерируют предложения Алгола 68. Другими словами, хотя Алгол 68 нельзя генерировать посредством контекстно-свободной грамматики, которая по определению может иметь только конечное множество правил, расширение (простое?) грамматики, разрешающее ей иметь бесконечное множество правил, служит достаточным обобщением, что позволяет специфицировать эту грамматику, генерирующую Алгол 68. В действительности с помощью такой грамматики можно генерировать любой язык типа 0. Это значит, что мы имеем дело с довольно мощной концепцией. Имея в виду тип распознавателя, который требуется языкам типа 0, эту концепцию можно считать даже слишком мощной для определения языков программирования, контекстно-зависимые средства которых носят довольно простой характер.

Применение второй грамматики дает возможность избежать рутинной работы, связанной с написанием бесконечного множества порождающих правил. Помимо прочего, формальное определение Алгола 68 содержит гиперправила, одно из которых (в слегка упрощенном виде) приводится ниже:

ref to MODE assignation:
ref to MODE destination, becomes token, MODE source. (1)

Здесь «:» употребляется вместо «→», а «,» — для разделения символов (терминалов или нетерминалов) в правой части гиперправила. В конце каждого правила ставится точка. Присваивание, получатель и источник имеют свой вид, и в соответствии с правилами W-грамматик слова, написанные заглавными буквами, должны *согласованно* замещаться при применении гиперправила. Гиперправило (в целом) представляет собой «скелет» бесконечного числа порождающих правил языка. Фактические порождающие правила грамматики получают путем *согласованной* подстановки метапонятий в гиперправиле с помощью терминальных порождающих метаправил, пользуясь метаправилами языка. Метаправила образуют контекстно-свободную грамматику, метапонятия которой являются нетерминалами, а терминальные метаправила (не содержащие заглавных букв) — терминалами. Заметим, что в этой грамматике отсутствует понятие начального символа.

В качестве примера рассмотрим (несколько упрощенное) метаправило:

MODE :: PLAIN; STOWED; ref to MODE; PROCEDURE; UNITED. (2)

В этом метаправиле «::» заменяет «:», а «;» используется (как в гиперправиле) для разделения вариантов.

Ни один из вариантов в (2) не является терминальным метаправилом, так что нельзя вывести порождающее правило Алгола 68 из (1), пользуясь только метаправилом (2). Однако другим метаправилом может быть следующее:

PLAIN :: INTREAL;boolean;character.

и у нас также есть

INTREAL::SIZETY integral; SIZETY real.

где одним из вариантов для *SIZETY* будет пустая строка.

Исследуя все возможные варианты, находим, что метапонятие *MODE* может генерировать бесконечное число терминальных метаправил в

соответствии с бесконечным числом видов в языке. (Например, *SIZETY* может генерировать любое число *LONG*).

Согласованная подстановка дает, например,

ref to real assignation:
ref to real destination, becomes token, real source.

что служит поэтому одним из бесконечного числа порождающих правил, которые можно вывести из (1). Заметим, однако, что

ref to real assignation:
ref to integral destination, becomes token, character source.

не является порождающим правилом языка из-за того, что не соблюдается правило согласованной подстановки.

Полный синтаксис Алгола 68 описывается с помощью двухуровневой грамматики. Двухуровневые грамматики используются также для описания семантики языков программирования [44].

Другой метод определения синтаксиса языка программирования — с помощью атрибутивной грамматики [34]. Симоне применил атрибутивную грамматику для описания подмножества Алгола 68 [51].

Рассмотрим назначение атрибутивных грамматик на примере одной из них, служащей для описания простого языка программирования. У этого языка такая же блочная структура, как у Алгола 60 или Алгола 68, и мы ассоциируем с каждым блоком список идентификаторов и их видов (или типов). Идентификаторы могут иметь один из двух видов **int** и **bool** и являются терминалами грамматики на этом уровне определения. Описания можно специфировать с помощью контекстно-свободных порождающих правил

dec→**int** *id*
dec→**bool** *id*

Здесь мы отступили от нашей обычной практики пользоваться строчными буквами только для терминалов.

Описав идентификатор, мы хотим запомнить его вид. Этот вид будет свойством описания, и можно видоизменить грамматику, чтобы это указать:

dec(.*ID*,*MODE*)→**int**(.*MODE1*)*id*(.*ID1*)
ID=*ID1*
MODE=*MODE1*

То же самое относится к описанию идентификатора вида **bool**. *MODE, MODE1, ID, ID1* пишутся в скобках после терминала или нетерминала грамматики и представляют собой его признаки, или *атрибуты*. С нетерминалом *dec* ассоциируются два атрибута; значение точки мы вскоре объясним. В соответствии с модифицированным контекстно-свободным порождающим правилом между атрибутами устанавливаются два соотношения. Таким образом, идентификатором, ассоциируемым с описанием, будет идентификатор, появляющийся после **int**, а видом, ассоциируемым с описанием,— вид **int**.

Теперь можем определить последовательность описаний;

decs (.*LIST*)→*dec* (.*ID, MODE*)
LIST=(*ID, MODE*)
decs (.*LIST*)→*decs* (.*LIST1*);*dec* (.*ID, MODE*)
LIST=*LIST1*+(*ID, MODE*)
ID is not contained in LIST1 (3)

Таким образом, *LIST* состоит из одной пары (*ID, MODE*) в случае одного описания (или первого описания) либо из последовательности идентификаторов и видов («+» означает конкатенацию). Ограничение

ID is not contained in LIST1

гарантирует, что ни один из идентификаторов не будет описан дважды в одном и том же блоке.

Теперь блок можно частично определить с помощью контекстно-свободного порождающего правила:

block→**begin** *decs; stats* **end**

Налагаем контекстно-зависимые требования:

block (ENV)→**begin** *decs (.LIST)*; *stats (ENV1)***end**
ENV1 = LIST + ENV

Языковой обстановкой, в которой выполняется *stats*, является среда самого блока, увеличенная на *decs*. Эта обстановка передается отдельным оператором

$$\begin{aligned} &stats\ (ENV) \rightarrow stat\ (ENV) \\ &stats\ (ENV) \rightarrow stats\ (ENV);\ stat\ (ENV) \end{aligned} \qquad (4)$$

и мы допускаем для упрощения, что *stat* принимает одну из форм:

stat (ENV)→*id (MODE1.ID1)* := *id (MODE2.ID2)*
MODE1 = search (ID1, ENV)
MODE2 = search (ID2, ENV)
MODE1 = MODE2

или

stat (ENV)→*block (ENV)*

Функция *search* будет искать обстановку снаружи, чтобы найти соответствующее описание идентификатора, а следовательно, и его вид. Если подходящее описание найти нельзя, выдается сообщение об ошибке.

Атрибуты применяются для описания контекстно-зависимых (или, правильнее, контекстно-несвободных) аспектов языка программирования. Их можно использовать для передачи информации из левой части порождающего правила в правую, как в (4). Такие атрибуты называются *унаследованными*. Атрибуты также можно использовать для передачи информации из правой части порождающего правила в левую, как в (3), тогда их называют *синтезированными атрибутами*. В приведенном выше примере в каждой паре скобок унаследованные атрибуты предшествуют синтезированным, элементы каждого списка разделяются запятыми, а два списка — точкой.

Преимущество атрибутивных грамматик заключается в том, что они выглядят как контекстно-свободные, но могут специфицировать контекстно-зависимые конструкции языка. Фактически любой язык типа 0 можно описать с помощью атрибутивной грамматики. Поскольку языки программирования представляются как контекстно-свободные, к которым добавляются контекстные ограничения, атрибутивные грамматики хорошо подходят для их описания. Существуют, кроме того, методы автоматического получения эффективных анализаторов из соответствующих атрибутивных грамматик. Этот факт, а также их удобочитаемость свидетельствуют о том, что атрибутивные грамматики близки к осуществлению двух основных назначений определения языка, упомянутых в начале данного раздела.

Среди других хорошо известных методов описания синтаксиса языков программирования следует упомянуть Vienna Definition Language для ПЛ/1 [42] и порождающие системы Ледгара [41].

2.4. ПРОБЛЕМА РАЗБОРА

Мы показали, как можно использовать грамматику для генерирования программ на заданном языке программирования. Однако компилятор должен решить не проблему генерирования программ, а проблему проверки строк символов, чтобы определить, принадлежат ли они данному языку, и если да, то распознать структуру строк в терминах порождающих правил грамматики. Эта проблема известна как *проблема разбора,* и, прежде чем приступить к ее рассмотрению, введем новые понятия.

Исследуем грамматику с порождающими правилами

$$\begin{array}{ll} 1.\ E \to E+T & 5.\ F \to (E) \\ 2.\ E \to T & 6.\ F \to x \\ 3.\ T \to T \times F & 7.\ F \to y \\ 4.\ T \to F & \end{array}$$

(E — начальный символ).
Ясно, что строка

$$(x+y) \times x$$

принадлежит данному языку. В частности, это можно вывести следующим образом:

$$\begin{array}{ll} E \Rightarrow T & \Rightarrow (F+T) \times F \\ \Rightarrow T \times F & \Rightarrow (x+T) \times F \\ \Rightarrow F \times F & \Rightarrow (x+F) \times F \\ \Rightarrow (E) \times F & \Rightarrow (x+y) \times F \\ \Rightarrow (E+T) \times F & \Rightarrow (x+y) \times x \\ \Rightarrow (T+T) \times F & \end{array}$$

Или же это можно было бы вывести так:

$$\begin{array}{ll} E \Rightarrow T & \Rightarrow (E+F) \times x \\ \Rightarrow T \times F & \Rightarrow (E+y) \times x \\ \Rightarrow T \times x & \Rightarrow (T+y) \times x \\ \Rightarrow F \times x & \Rightarrow (F+y) \times x \\ \Rightarrow (E) \times x & \Rightarrow (x+y) \times x \\ \Rightarrow (E+T) \times x & \end{array}$$

Отметим, что на каждом этапе первого вывода самый левый нетерминал в сентенциальной форме замещается с помощью одного из порождающих правил грамматики. Поэтому данный вывод называется *левосторонним.* Второй вывод, на каждом этапе которого замещается самый правый нетерминал, называется *правосторонним.* Существуют также другие выводы, не являющиеся ни лево-, ни правосторонними.

Мы определяем *левосторонний разбор* предложения как последовательность порождающих правил, применяемую для генерирования предложения посредством левостороннего вывода. В данном случае левосторонний разбор можно записать как

2, 3, 4, 5, 1, 2, 4, 6, 4, 7, 6.

Правосторонний разбор предложения является обратной последовательностью порождающих правил, используемых для генерирования

предложения посредством правостороннего вывода; например, в вышеприведенном случае правосторонний разбор запишется в виде

$$6, 4, 2, 7, 4, 1, 5, 4, 6, 3, 2.$$

Обратный порядок последовательности порождающих правил связан с тем, что правосторонний разбор обычно ассоциируется с приведением предложения к начальному символу, а не с генерированием предложения из начального символа (см. ниже разбор снизу вверх). Заметим, что каждое порождающее правило используется в обоих выводах (или разборах) одинаковое число раз.

Можем также описать вывод и в терминах построения дерева, известного как синтаксическое дерево (или дерево разбора). В случае со строкой

$$(x+y)\times x$$

синтаксическое дерево будет таким, как показано на рис. 2.1. Проблему разбора можно свести к

1) нахождению левостороннего разбора;
2) нахождению правостороннего разбора;
3) построению синтаксического дерева.

В большинстве случаев левосторонний и правосторонний разборы и синтаксическое дерево являются уникальными. Однако рассмотрим грамматику с порождающими правилами:

$$S \to S+S$$
$$S \to x$$

Предложение $x+x+x$ имеет два синтаксических дерева (рис. 2.2) и два левосторонних (и правосторонних) разбора:

$$S \underset{L}{\Rightarrow} S+S \underset{L}{\Rightarrow} S+S+S \underset{L}{\Rightarrow} x+S+S \underset{L}{\Rightarrow} x+x+S \underset{L}{\Rightarrow} x+x+x$$

$$S \underset{L}{\Rightarrow} S+S \underset{L}{\Rightarrow} x+S \underset{L}{\Rightarrow} x+S+S \underset{L}{\Rightarrow} x+x+S \underset{L}{\Rightarrow} x+x+x$$

где $\underset{L}{\Rightarrow}$ означает «выводится левосторонним выводом».

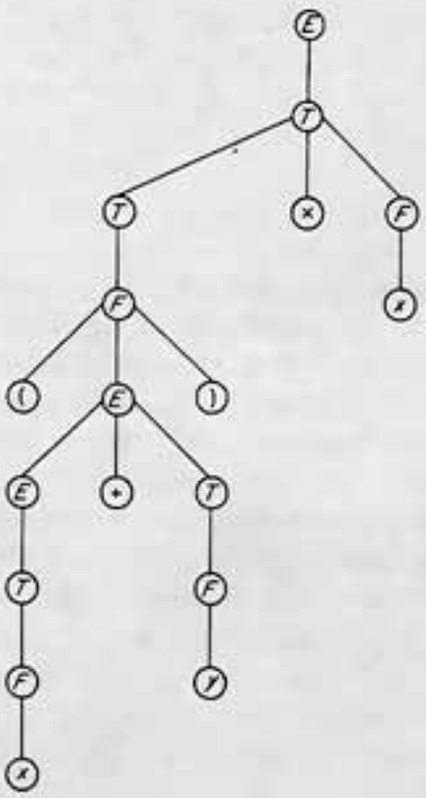


Рис. 2.1

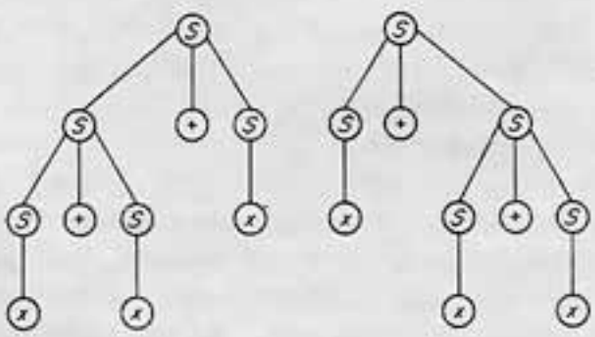


Рис. 2.2

Если какое-либо предложение, генерированное грамматикой, имеет более одного дерева разбора, о такой грамматике говорят, что она неоднозначна. Эквивалентное условие заключается в том, что предложение должно иметь более одного левостороннего или правостороннего разбора. Задача установления неоднозначности какой-либо грамматики является неразрешимой. Иными словами, не существует алгоритма, который принимал бы *любую* грамматику в качестве входа и определял бы, однозначна она или нет (хотя можно обнаружить такие особые случаи, как порождающие правила, которые содержат и левую, и правую рекурсии). В некоторые языки уже заложена неоднозначность, т. е. их нельзя генерировать с помощью однозначной грамматики. С другой стороны, некоторые неоднозначные грамматики можно преобразовать в однозначные, генерирующие тот же язык. Например, грамматика с порождающими правилами

$$S \to x$$
$$S \to S + x$$

является однозначной и генерирует тот же язык, что и рассмотренная ранее неоднозначная грамматика. Для большинства методов разбора требуется однозначная грамматика. Если существует алгоритм, определяющий пригодность заданной грамматики для конкретного метода разбора, и этот метод можно применять только для подмножества однозначных грамматик, то мы имеем алгоритм, которым можно воспользоваться для того, чтобы показать, что грамматика является однозначной (но не для того, чтобы показать, что она является неоднозначной).

Де Ремер [17] сравнил задачу разбора с игрой в домино. В этой игре для каждого порождающего правила грамматики выделяется пластинка, верхняя и нижняя части которой соответствуют левым и правым сторонам правила. Для грамматики, рассмотренной в начале данной главы, пластинки окажутся такими, как показано на рис. 2.3. Игру

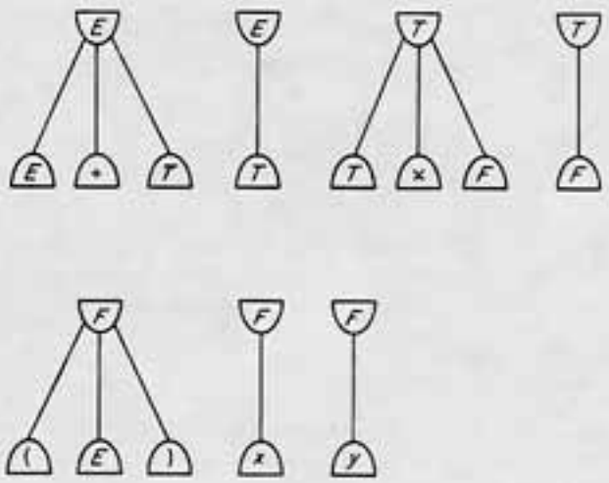


Рис. 2.3

начинаем на большой доске. Начальный символ, заключенный в полукруг с плоским основанием, находится в верхней части доски, а символы в предложении, заключенные в полукруг с плоским верхом, располагаются в ряд в нижней части доски. Например, для разбора предложения

$$(x+y)\times x$$

начальная позиция представлена на рис. 2.4. Цель игры заключается в том, чтобы с помощью пластинок соединить предложение и начальный символ. Здесь необходимо подчеркнуть следующие положения:

1. Существует столько копий каждой пластинки, сколько требуется.
2. Пластинки гибкие, и их можно по потребности вытягивать.

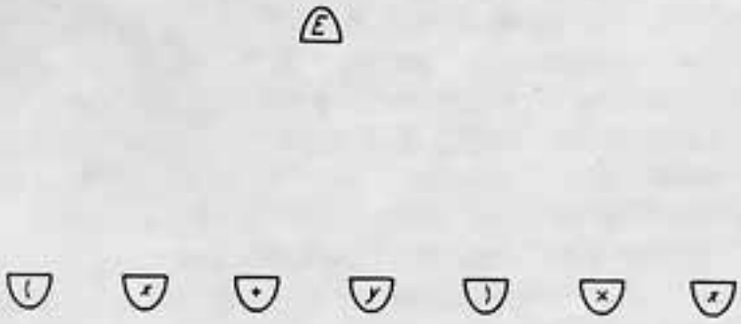


Рис. 2.4

3. Нельзя играть перевернутыми пластинками.
4. Закругленные сверху и снизу пластинки, обозначающие один и тот же символ, соединяются по плоским граням.

Решение вышеописанного примера приведено на рис. 2.5.

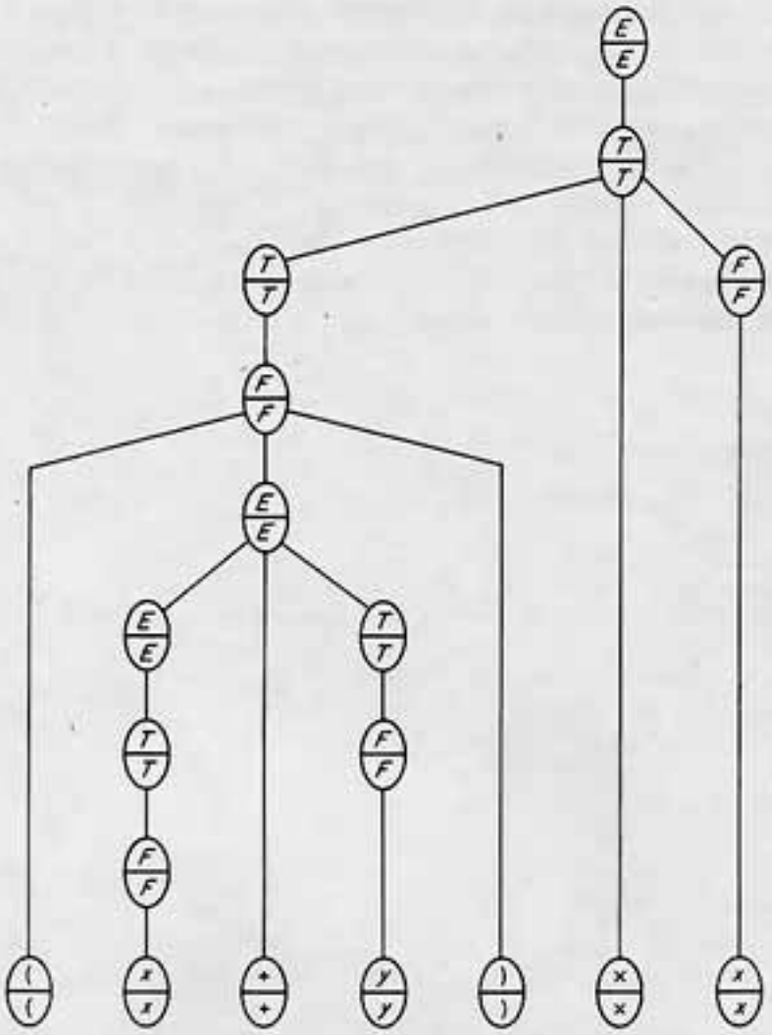


Рис. 2.5

Существует несколько способов решения задачи игры в домино. Можно начать с верхней части доски с начального символа и пытаться пройти вниз к предложению или же начать с предложения и попытать-

ся пройти наверх к начальному символу. Допускается начать с левого края доски и идти к правому или наоборот, либо от верхнего левого угла идти к нижнему правому углу доски. Вполне естественно желание решать задачу, не передвигая однажды поставленную пластинку, или же отказаться от какой-либо части решения и начать сначала.

Большинство этих методов решения задачи соответствует конкретным методам разбора. Методы разбора обычно бывают нисходящими, т. е. начинают с начального символа и идут к предложению, или восходящими, т. е. начинают с предложения и идут к начальному символу. Разрешается смешивать эти два метода. Предложение, разбираемое слева направо, читается нормально, хотя с точки зрения разбора читать его справа налево может оказаться так же просто (или еще проще). Принцип никогда не поднимать уже положенную пластинку в игре такого типа соответствует принципу никогда не отказываться от решения применять конкретное порождающее правило в процессе разбора. Отказ от решения в разборе иногда называют возвратом. Методы разбора могут быть недетерминированными или детерминированными в зависимости от того, возможен возврат или нет. Недетерминированные методы разбора весьма дорогие с точки зрения памяти и времени и крайне затрудняют включение в синтаксический анализатор действий, выполняемых во время компиляции, результаты которых позднее должны быть аннулированы (например, построение таблицы символов и т. п.). В настоящей книге мы имеем дело почти исключительно с детерминированными методами разбора. В гл. 4 описывается метод LL, который осуществляет разбор сверху вниз и использует левосторонние выводы, а в гл. 5 — метод LR, который осуществляет разбор снизу вверх и использует правосторонние выводы. Мы в основном рассматриваем контекстно-свободный разбор, но эти методы можно распространить и на определенные классы универсальных грамматик, таких, как атрибутивные [57] и аффиксные [40]. W-грамматики менее удобны для разбора из-за своего более общего характера.

Упражнения

2.1. Приведите грамматику для следующих языков:

$$L_1 = \{a^m b^n c^p \mid m,\ n,\ p \geqslant 0\},$$
$$L_2 = \{a^m b^n c^p \mid m,\ p \geqslant 0\},$$
$$L_3 = \{x^n a y^n \mid n \geqslant 0\} \cup \{x^n b y^n \mid n \geqslant 0\}.$$

2.2. Является ли какой-либо из этих языков регулярным? Обоснуйте свой вывод.

2.3. Найдите регулярную грамматику, генерирующую тот же язык, что и грамматика со следующими порождающими правилами (S — начальный символ):

$$S \to AB$$
$$A \to X \mid Y$$
$$X \to x \mid xX$$
$$Y \to y \mid yY$$
$$B \to b \mid bB$$

2.4. Выпишите регулярное выражение, генерированное грамматикой из упражнения 2.3.

2.5. Покажите, что грамматика со следующими порождающими правилами является неоднозначной:

$$S \to \textbf{if } b \textbf{ then } S \textbf{ else } S$$
$$\to \textbf{if } b \textbf{ then } S$$
$$\to a$$

2.6. В некоторой грамматике S является начальным символом, а порождающие правила следующие:

$$S \rightarrow \textbf{real}\ IDLIST$$
$$IDLIST \rightarrow IDLIST,\ ID$$
$$IDLIST \rightarrow ID$$
$$ID \rightarrow a$$
$$ID \rightarrow b$$
$$ID \rightarrow c$$

Составьте дерево разбора для предложения

real *a, b, c*

2.7. Выведите грамматику, генерирующую все строки (включая пустую) по алфавиту {0, 1}.

2.8. Рассмотрите следующую грамматику с начальным символом *PROGRAM*:

$$PROGRAM \rightarrow \textbf{begin}\ DECS;\ STATS\ \textbf{end}$$
$$DECS \rightarrow d;\ DECS$$
$$d$$
$$STATS \rightarrow s;\ STATS$$
$$s$$

Дайте левосторонний и правосторонний выводы

begin *s; s; d; d* **end**

2.9. Идентификатор в Фортране состоит из последовательности, включающей до шести букв и цифр, первая литера из которых должна быть буквой. Выведите:

а) регулярное выражение для идентификатора;

б) грамматику типа 3, генерирующую все без исключения идентификаторы Фортрана.

2.10. В большинстве языков имеется условное предложение или оператор, аналогичный описанному в упражнении 2.5. Объясните, как в некоторых знакомых вам языках избегают потенциальной неоднозначности или как там разрешается этот вопрос.

Глава 3

ЛЕКСИЧЕСКИЙ АНАЛИЗ

Первой фазой процесса компиляции по идее является лексический анализ, т. е. группирование строк литер, обозначающих идентификаторы, константы или слова языка и т. д., в единые символы. Конечно, в однопроходных компиляторах этот процесс идет параллельно с другими фазами компиляции. Однако независимо от того, составляет ли лексический анализ отдельную фазу компиляции, при описании конструкции компилятора, а также его построении удобно представлять себе лексический анализ как самостоятельную фазу. В настоящей главе мы рассмотрим, как распознаются различные символы, и опишем таблицы, которые лексический анализатор должен построить для себя и для последующих фаз компилятора. Обсудим также конкретные проблемы, встречающиеся при написании лексических анализаторов для Алгола 68 и других языков.

3.1. РАСПОЗНАВАНИЕ СИМВОЛОВ

Характер распознаваемых строк может намного упростить процесс лексического анализа. К примеру, возьмем такие идентификаторы, как

first *date1* *no1234*,

вещественные числа

1.23 *6.0* $34.29_{10}-20$,

слова языка

begin **for** **case**

Все эти строки можно генерировать с помощью регулярных выражений. Например, вещественные числа можно генерировать посредством регулярного выражения

$$(+|-|)\ digit^*.digit\ digit^*({}_{10}(+|-|)\ digit\ digit^*|)$$

из которого видно, что реальное число состоит из следующих компонентов, расположенных именно в таком порядке:

1) возможного знака;
2) последовательности из нуля или более цифр;
3) десятичной точки;
4) последовательности из одной или более цифр;
5) возможной экспонентной части, содержащей
   а) экспоненту $({}_{10})$,
   б) возможный знак,
   в) последовательность из одной или более цифр.

Регулярные выражения, конечно, эквивалентны грамматикам типа 3 (см. разд. 2.2). Например, грамматика типа 3, соответствующая регулярному выражению для вышеприведенного вещественного числа, имеет порождающие правила:

$$
\begin{array}{l}
R \to +S \mid \\
\quad -S \mid \\
\quad dS \mid \\
\quad .P \\
S \to dS \mid \\
\quad .P \\
P \to dF \mid \\
\quad d \\
F \to dF \mid \\
\quad {}_{10}A \mid \\
\quad d \\
A \to +N \mid \\
\quad -N \mid \\
\quad dN \mid \\
\quad d \\
N \to dN \mid \\
\quad d
\end{array}
$$

где R — начальный символ.

Существует полное соответствие между регулярными выражениями (а поэтому и грамматиками типа 3) и *конечными автоматами*, которые определяются следующим образом.

Конечный автомат — это устройство для распознавания строк какого-либо языка. У него есть конечное множество состояний, отдельные из которых называются последними. По мере считывания каждой литеры строки контроль передается от состояния к состоянию в соответствии с заданным множеством переходов. Если после считывания последней литеры строки автомат будет находиться в одном из последних состояний, о строке говорят, что она принадлежит языку, принимаемому автоматом. В ином случае строка не принадлежит языку, принимаемому автоматом.

Формально конечный автомат определяется пятью характеристиками:

1) конечным множеством состояний K;
2) конечным входным алфавитом Σ;
3) множеством переходов δ;
4) начальным состоянием $S(S \in K)$;
5) множеством последних состояний $F(F \subseteq K)$.

Конечный автомат можно записать в виде пятерки

$$M = (K, \Sigma, \delta, S, F).$$

Рассмотрим следующий пример, где состояниями являются A и B, входным алфавитом — {0, 1}, начальным состоянием — A, множеством последних состояний — $\{A\}$, а переходами

$$
\begin{array}{l}
\delta(A,0) = A, \\
\delta(A,1) = B, \\
\delta(B,0) = B, \\
\delta(B,1) = A.
\end{array}
$$

Эти переходы означают, что «при чтении *0* в состоянии A управление передается в состояние A» и т. д.

При чтении строки

01001011

управление последовательно передается в следующем порядке через состояния

$$A, A, B, B, B, A, A, B, A.$$

Так как A есть последнее состояние, строка принимается конечным автоматом. Однако при чтении строки

$$00111$$

автомат проходит через состояния

$$A, A, A, B, A, B$$

в таком порядке, и поскольку B не является последним состоянием, эта строка не принимается, т. е. она не принадлежит языку, принимаемому этим автоматом. В связи с тем что нули не влияют на состояние автомата, а каждая единица изменяет его состояние с A на B и с B на A, и начальное состояние является тем же, что и последнее состояние, язык, принимаемый автоматом, состоит из тех строк, которые содержат четное число единиц.

Переходы можно представить также с помощью таблицы (табл. 3.1) и схематически (рис. 3.1).

Таблица 3.1

Вход	Состояния A	Состояния B
0	A	B
1	B	A

Рис. 3.1

Такой автомат называют детерминированным, потому что в каждом элементе таблицы переходов содержится одно состояние. В недетерминированном конечном автомате это положение не выдерживается. Рассмотрим конечный автомат, определенный следующим образом:

$$M_1=(K_1,\Sigma_1,\delta_1,S_1,F_1),$$

где $K_1=\{A,B\}$, $\Sigma_1=\{0,1\}$, $S_1=\{A\}$, $F_1=\{B\}$, а переходы представлены в табл. 3.2 или схематически на рис. 3.2. M_1 принимает строку тогда и только тогда, когда она начинается с единицы. Эта же строка не может быть принята, так как при чтении 0 не осуществляется переход с A.

Таблица 3.2

	A	B
0	∅	{B}
1	{A, B}	{B}

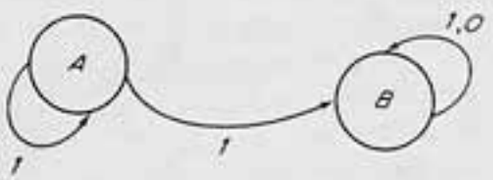


Рис. 3.2

Строка 1 будет принята: здесь есть переход (в более общем случае последовательность переходов), ведущий к последнему состоянию при чтении строки. Имеется также переход к непоследнему состоянию (от A к A), но это не влияет на приемлемость строки. Поэтому прежде чем прийти к выводу о том, что строка не может быть принята недетерминированным конечным автоматом, необходимо перепробовать все

возможные последовательности переходов. С этой целью следует перебрать все возможности одну за другой или параллельно. Если в данный момент времени можно проходить только по одному пути (как обычно и бывает), то требуется возвратиться назад, т. е. необходимо перечитать всю строку или ее часть в процессе опробования другого пути. Очевидно, что эта процедура может оказаться весьма длительной.

Существует детерминированный конечный автомат M_2, соответствующий автомату M_1, который принимает тот же язык,

$$M_2=(K_2,\Sigma_2,\delta_2,S_2,F_2),$$

где $K_2=\{A, B, C\}$, $\Sigma_2=\{0, 1\}$, $S_2=\{A\}$, $F_2=\{B\}$, а переходы даны в табл. 3.3 и схематически на рис. 3.3.

Таблица 3.3

	A	B	C
0	C	B	C
1	B	B	C

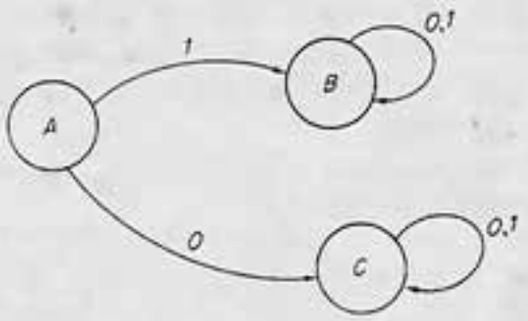


Рис. 3.3

Как и M_1, M_2 принимает строку из нулей и единиц тогда и только тогда, когда строка начинается с единицы. Однако при распознавании строки с помощью M_2 возврат никогда не требуется, так как в процессе чтения определенного входного символа из любого состояния произойдет точно один переход к другому состоянию. Это значит, что при использовании M_2 время распознавания строки будет пропорционально ее длине.

Можно доказать, что каждому недетерминированному конечному автомату соответствует детерминированный конечный автомат, принимающий тот же язык. В доказательстве допускается, что состояние в детерминированном автомате соответствует подмножествам состояний недетерминированного автомата. Например, состояние B в M_2 соответствует множеству состояний $\{A, B\}$ в M_1, а состояние C в M_2 — $\varnothing$, пустому множеству состояний в M_1.

Соответствие лексическому анализу заключается в том, что каждому языку типа 3 соответствует детерминированный конечный автомат, который распознает строки этого языка. Например, строки, генерируемые грамматикой G_1 с порождающими правилами

$A\to 1A$ $\quad$ $B\to 1B$
$A\to 1B$ $\quad$ $B\to 0$
$A\to 1$ $\quad$ $B\to 11$
$B\to 0B$

где A — начальный символ, распознаются с помощью M_1 или M_2. Грамматику получают из недетерминированного конечного автомата M_1 следующим образом:

1. Начальное состояние автомата становится начальным символом предложения грамматики.

2. Переходам

$$\delta(A,\ 1)=A,$$
$$\delta(A,\ 1)=B,$$
$$\delta(B,\ 0)=B,$$
$$\delta(B,\ 1)=B$$

соответствуют порождающие правила:

$$A\to 1A$$
$$A\to 1B$$
$$B\to 0B$$
$$B\to 1B$$

а тому, что в состоянии A есть переход при чтении 1 к последнему состоянию (B), соответствует

$$A\to 1$$

и аналогично

$$B\to 1$$
$$B\to 0$$

Можно также, наоборот, из грамматики вывести автомат M_1. Автомат M_2 соответствует грамматике G_2 с порождающими правилами

$A\to 1B$
$A\to 0C$
$A\to 1$
$B\to 0B$
$B\to 1B$
$B\to 0$
$B\to 1$
$C\to 0C$
$C\to 1C$

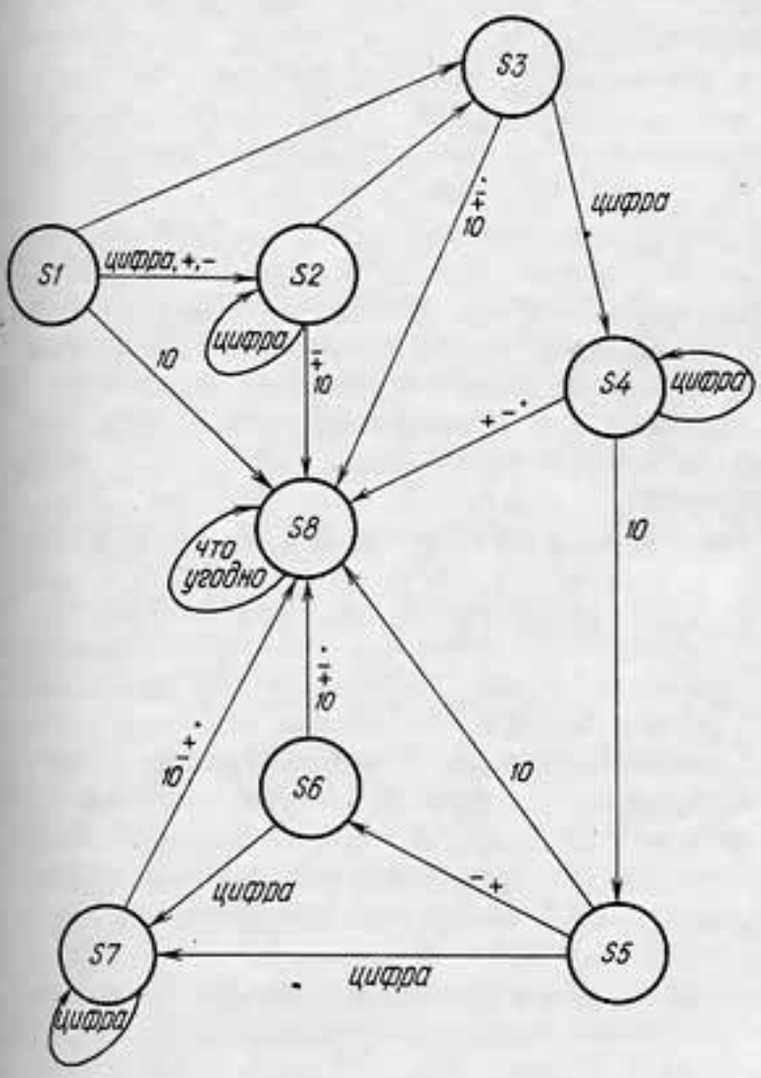


Рис. 3.4

которая содержит бесполезный нетерминал C, но эквивалентна G_1. Грамматику G_2 называют «нечистой», так как она включает нетерминал, не генерирующий никаких терминальных символов.

Написание лексического анализатора заключается частично в моделировании различных автоматов для распознавания идентификаторов, чисел, зарезервированных слов и т. д. Конечный автомат для распознавания вещественного числа показан на рис. 3.4. $S1$ является начальным состоянием, а $S4$ и $S7$ — последними состояниями; $S8$ — состояние отказа. Этот автомат — детерминированный, и его можно получить из грамматики типа 3, описанной в начале раздела, в два этапа. Сначала выводится недетерминированный автомат, состояния которого (кроме последнего) соответствуют нетерминалам грамматики, а затем его делают детерминированным.

На основе этого автомата можно написать процедуру для распознавания вещественного числа. Процедура дает значения **true** или **false** в зависимости от того, распознается вещественное число или нет.

```
proc realno = bool:
begin char in; int i := 1;
  while read (in); digit (in) or
  sign (in) or in = "." or in = "10"
  do
  i := case i in
  if sign (in) or digit (in) then 2
  elif in = "." then 3
  else 8 fi,
  if digit (in) then 2
  elif in = "." then 3
  else 8 fi,
  if digit (in) then 4
  else 8 fi,
  if digit (in) then 4
  elif in = "10" then 5
  else 8 fi,
  if digit (in) then 7
  elif sign (in) then 6
  else 8 fi,
  if digit (in) then 7
  else 8 fi,
  if digit (in) then 7
  else 8 fi,
  8 esac
  od;
  backspace (standin);
  i = 4 or i = 7
end
```

Допускается существование процедуры *digit*, проверяющей литеру, не является ли она цифрой, и процедуры *sign*, проверяющей наличие «+» или «—». Процедура заканчивается, когда она доходит до литеры, которая не может содержаться в вещественном числе. Эта последняя литера, считающаяся началом какого-либо другого символа, тогда «не читается».

Если два символа начинаются с одной и той же литеры или строки литер, может оказаться необходимым «соединение» процедур их распознавания.

Мы показали, что, имея конечный автомат, принимающий строки регулярного языка, нетрудно написать процедуру для распознавания этих строк. Конечно, каждому конечному автомату соответствует регулярное выражение, которое было ранее дано для вещественного числа как

$$(+|-|)\ digit^{*}\ .\ digit\ digit^{*}\ ({}_{10}(+|-|)\ digit\ digit^{*}|)$$

На основе регулярного выражения можно написать процедуру для распознавания вещественного числа

```
proc realno1 = void:
begin char in; read (in);
  if sign (in) then read (in) f
  while digit (in)
  do read (in)
  od;
  if in = "." then read (in)
     else error fi;
  if digit (in) then read (in)
     else error fi;
  while digit (in)
  do
     read (in)
  od;
  if in = "10" then read (in);
     if sign (in) then read (in) fi;
     if digit (in) then read (in) else error fi;
     while digit (in)
     do read (in)
     od
  fi;
  backspace (standin)
end
```

error — это процедура, которая, как предполагается, принимает соответствующие меры, когда возникает ошибка.

Распознаватели можно создавать с помощью регулярных выражений или конечных автоматов почти так же быстро, как мы пишем, и этот процесс нетрудно автоматизировать.

3.2. ВЫХОД ИЗ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

Лексический анализатор преобразует исходную программу в последовательность символов. При этом идентификаторы и константы, имеющие произвольную длину, заменяются символами фиксированной длины. Слова языка также заменяются каким-нибудь стандартным представлением. Например, слова языка могут заменяться целыми числами, идентификаторы — буквой *I* и следующим за ней целым числом, константы — буквой *C* с последующим целым числом и т. д. Эти целые числа не могут, конечно, иметь произвольную длину, и поэтому число идентификаторов и констант, появляющихся в программе, обычно ограничивают величиной, не превышающей какого-либо довольно большого целого числа. Большинству пользователей такое ограничение не должно быть заметно. Однако если этот предел когда-нибудь достигается, компилятор должен выдать четкое сообщение.

Коды, создаваемые для слов языка, не зависят от программы. Например, **begin** может быть заменено на *11*, а **for** — на *22*. Однако стандартные идентификаторы, такие, как *read, print, pi,* всегда должны иметь одинаковые коды, а идентификаторам, определяемым пользователем, код выдается лексическим анализатором в том порядке, в котором они встречаются. Например, первым идентификатором будет *11*, вторым — *12* и т. д. Каждый экземпляр определенного идентификатора (на любом уровне блока) заменяется одним и тем же кодом. Это значит, что лексический анализатор должен вести таблицу идентификаторов, дающую коды всем идентификаторам. После осуществления лексического анализа каждый идентификатор согласованно замещается, и, строго говоря, отпадает необходимость в таблице идентификаторов. Однако она может пригодиться пользователям при выдаче диагностики в терминах исходного языка, если на последующей стадии компиляции обнаружатся ошибки. Поэтому таблицу идентификаторов обычно сохраняют в течение всего процесса компиляции.

Для хранения таблицы идентификаторов в памяти обычно требуется структура данных, которая допускает включение идентификаторов произвольной длины. Очевидное решение — применение списков, где каждый элемент соответствует букве идентификатора, а ☑ представляет нулевой указатель (рис. 3.5).

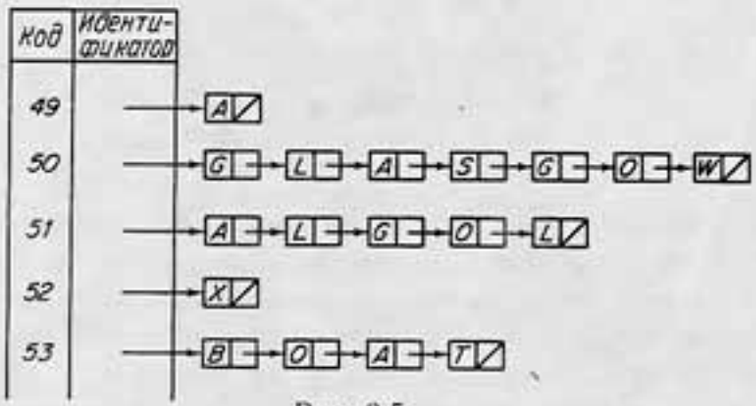


Рис. 3.5

В Алголе 68 для таблицы идентификаторов применяются описания вида

```
mode list = struct (char l, ref list next);
mode idtable = [1 : p] struct (int code, ref list ident)
```

Большое число указателей в этой структуре ведет к излишней трате объема памяти. Другой вариант представления букв идентификатора заключается в использовании строки литер или ссылки на строку литер, например

```
mode idtable1 = [1 : p] struct (int code, ref[ ] char ident)
```

В любом случае сама таблица идентификаторов должна быть расширяемой. Для этого можно сделать таблицу списком, но лучшим решением будет, вероятно, применение гибкого массива. Однако вместо того, чтобы расширять массив поэлементно, эффективнее начать с массива, состоящего, скажем, из 50 элементов, и добавлять по 10 или 20 элементов всякий раз, когда его нужно расширить. Эффект гибкости в Алголе 68 (и, возможно, в других языках) может быть также обес-

печен введением ссылки на вид строки и помещением копий элементов в бóльшую строку в тех ситуациях, когда таблицу необходимо расширять.

Несколько иной подход к реализации таблицы идентификаторов заключается в том, что все литеры, образующие различные идентификаторы, хранятся в едином массиве литер, а в каждой записи в таблице содержится указатель на начало строки литер, соответствующих конкретному идентификатору, и счетчик числа содержащихся в ней литер. Чтобы проверить, находится ли какой-либо определенный идентификатор в таблице, нужно осуществить только политерное сравнение с теми идентификаторами, уже находящимися в таблице, которые имеют соответствующую длину (рис. 3.6).

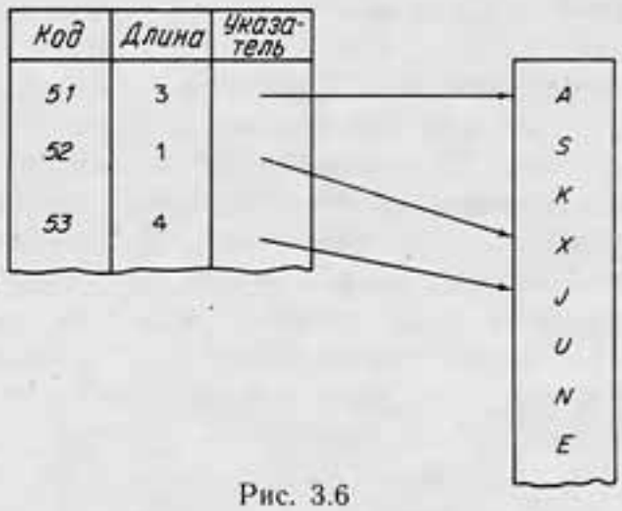


Рис. 3.6

Аналогичная таблица нужна и для констант. Некоторые компиляторы проверяют длину констант и вычисляют их в процессе лексического анализа. Однако допустимая длина константы зависит обычно от основной машины, и лучше просто держать константы произвольной длины в таблице во время лексического анализа, чем вводить зависимость от машины на такой ранней стадии процесса компиляции. Таблица констант потребуется на поздней стадии компиляции, а лексический анализатор должен передать синтаксическому анализатору лишь то, что константа определенного типа хранится в соответствующей таблице по указанному адресу. Значение, хранящееся по этому адресу, может не понадобиться до последней фазы генерирования кода, но его тип должен быть известен на стадии анализа. Лексический анализатор передаст синтаксическому анализатору код в виде *TXXX* (*XXX* обозначает запись в таблице констант, где хранится значение, а *T* — его тип). «Небольшие» целочисленные константы можно передавать в таблицу констант как литералы, а не указатели. Например, *C001 C009* могут представлять целые числа от *1* до *9*.

В программах, написанных на Алголе 68, могут содержаться определяемые пользователем выделенные идентификаторы (или выделенные слова), такие, как

```
list max
```

которые служат для спецификации видов или обозначения операций. Лексический анализатор обращается с ними, как с идентификаторами, помещая их в таблицу, используемую дальнейшими проходами для диагностических целей.

3.3. КОММЕНТАРИИ И Т. Д.

Лексический анализатор наряду с передачей представления символов в исходной программе может включать в исходный текст дополнительную информацию или исключать из него строки литер. Примером такой дополнительной информации служат номера строк. В диагностике, особенно для многопроходного компилятора, полезно иметь возможность ссылаться на ошибки в программе с помощью номеров строк оригинального исходного текста. Для лексического анализатора не составляет труда включать в программу по мере ее обработки номера строк, которые могут передаваться последующим проходам до тех пор, пока существует возможность обнаружения ошибок (даже до стадии генерирования машинного кода, если константы не проверяются до этой стадии).

Комментарии включаются в программу только для читателя — человека. Компилятору до них нет никакого дела, и лексический анализатор их обычно удаляет. Программы на Алголе 68 могут также включать прагматические замечания, которые подобно комментариям не изменяют значения программ. Однако для компилятора прагматические замечания представляют интерес и могут повлиять на способ осуществления компиляции. В этих замечаниях, как правило, передаются сообщения от программиста компилятору относительно, например, того, какую конкретную часть программы нужно оптимизировать по времени ее исполнения или какой конкретный метод исправления ошибок следует использовать. Прагматические замечания могут относиться к одной конкретной фазе компилятора, такой, как лексический анализатор или генератор кода, либо более чем к одной. Если прагматическое замечание касается только лексического анализатора, оно удаляется на этой стадии. В ином случае оно передается дальше.

3.4. ПРОБЛЕМЫ, СВЯЗАННЫЕ С КОНКРЕТНЫМИ ЯЗЫКАМИ

Лексический анализатор в основном базирует свои решения на информации, содержащейся в символе, который он обрабатывает. Обычно ожидают, что лексический анализатор примет любую последовательность символов, принадлежащих компилируемому языку. Например, при распознавании целочисленной константы для лексического анализатора не важно, следует ли она за оператором либо за символом **begin** или находится в объявлении процедуры либо в операторе **do**. Если способ объединения литер в символы зависит от контекста, может возникнуть затруднение. Например, в Фортране в последовательности

DO 7 I=1, 5

«*DO*» до момента чтения запятой может быть словом языка или же частью идентификатора

DO 7 I

Чтобы разрешить неоднозначность, лексический анализатор должен немного заглянуть вперед. Проблема возникает потому, что в Фортране пробелы не имеют значения. В ПЛ/1 слова языка (в основном) не являются зарезервированными, и для разрешения некоторых неоднозначностей может потребоваться предварительный просмотр. В Коболе

слова языка (их свыше 100) зарезервированы, поэтому они не используются в качестве идентификаторов. В языках Бейсик и Алгол 68 слова принимают форму, отличную от идентификаторов, и их нельзя спутать.

Однако в Алголе 68 возникает другая проблема. Если допустить, что знаки операций являются символами языка, то последовательность литер $<=$ может быть интерпретирована как знак операции (обычно означающий «меньше или равно»). Однако такая же последовательность литер может встретиться и в программе на Алголе 68 в описании знака операции $<$, например

$$\mathbf{op} <= \cdots$$

В этом случае «$<$» есть знак операции, а «$=$» — символ, отделяющий имя знака операции от определяющей ее процедуры.

От лексического анализатора в такой ситуации ожидается передача «$<$» и «$=$» как двух разных символов. Но для того чтобы этот анализатор смог различить сложный символ $<=$ и два символа «$<$» и «$=$», следующие друг за другом, ему необходимо иметь некоторую контекстуальную информацию. Очевидно, если **op** предшествует «$<=$», их нужно рассматривать как два различных символа. В приведенном ниже контексте, где у нас имеется последовательность описаний знаков операций

$$\mathbf{op} >= \cdots, <=$$

разделенных запятыми, «$<$» и «$=$» также представляют собой разные символы. Чтобы справиться с подобными ситуациями, лексическому анализатору Алгола 68 иногда должно быть известно (по крайней мере, во избежание осложнений на последующем этапе) что-либо о контексте, в котором он работает.

Источником затруднений в Алголе 68 служат также форматы. Внутри формата (используемого для ввода-вывода) некоторые буквы, например *X*, *Y*, *L*, в одних местах имеют точно определенное значение, а в других их можно принять за идентификаторы (или части идентификаторов). Каждый формат заключен в пару символов формата (обычно это знаки доллара), так что лексический анализатор всегда знает, находится ли он внутри формата или нет, и может интерпретировать буквы *X*, *Y*, *L* и т. д. соответствующим образом. Осложнение возникает в тех форматах, которые содержат замкнутые предложения (блоки), где для этих литер нужно, конечно, опять вводить нормальную интерпретацию. В

$$\$n(x-1)x,\ dd\$$$

перед написанием (предполагается, что формат используется в связи с выводом) целого числа, состоящего из двух цифр, выделяются $x-1$ пробелов (второй x является символом размещения, обозначающим пробел).

Замкнутое предложение внутри формата может также содержать формат и т. д. до любой глубины вложения. В этом случае лексический анализатор должен состоять из двух взаимно рекурсивных процедур, соответствующих каждому из тех видов, в которых ему предстоит работать.

Мы уже упоминали о том, что в программе могут содержаться определяемые пользователем выделенные слова, обозначающие виды или знаки операций. Однако лексический анализатор не имеет контекстуаль-

ной информации, и выделенные слова просто передаются как таковые. В каждой реализации Алгола 68 есть метод представления выделенных слов (и слов языка) в программе. Приведем два наиболее общих представления слова языка **begin**:

'BEGIN' и *.BEGIN*

Можно построить лексический анализатор таким образом, чтобы он принимал более одного представления выделенных символов, причем будет допускаться одно конкретное представление, если программист не даст другого указания с помощью прагматического замечания. Стандартное представление для Алгола 68 описано в [25].

Проблемы, рассмотренные в настоящем разделе, существенно усложняют создание лексических анализаторов. Без этих проблем лексические анализаторы можно было бы создавать автоматически с помощью специального генератора, принимающего на входе синтаксис, который состоит из различных символов конкретного языка, и выдающего лексический анализатор для этого языка.

Процесс лексического анализа может быть достаточно простым, но в смысле времени компиляции он оказывается довольно дорогим. Больше половины времени, затрачиваемого компиляторами на компиляцию программы, приходится на лексический анализ (в основном из-за литерного характера ввода). Если бы для какого-либо компилятора на это потребовалось меньшее время, то мы могли бы предположить, что какие-то другие фазы компилятора были не в полной мере эффективными.

Упражнения

3.1. Запишите порождающие правила грамматики типа 3, генерирующей регулярное выражение

$$(101)^* (010)^*$$

3.2. Выведите конечный автомат, соответствующий регулярному выражению в упражнении 3.1.

3.3. Выведите недетерминированный конечный автомат, принимающий регулярное выражение

$$(101)^* (110)^*$$

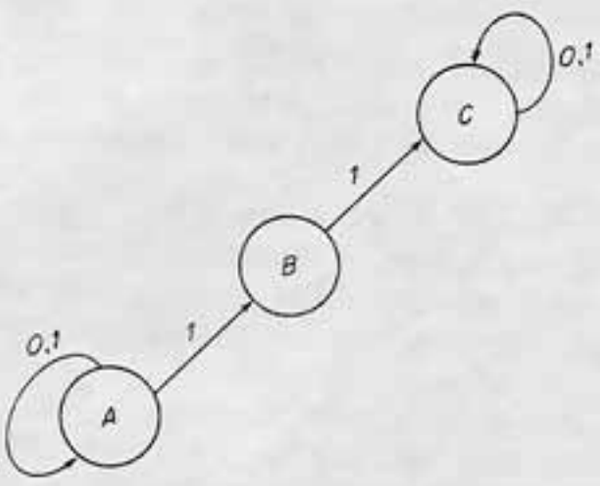


Рис. 3.7

3.4. Выведите детерминированный автомат, принимающий регулярное выражение из упражнения 3.3.

3.5. Приведите грамматику типа 3, соответствующую конечному автомату, представленному на рис. 3.7, где A есть начальное состояние, а C — последнее. Как бы вы описали этот язык?

3.6. Напишите процедуру распознавания строк языка, принимаемого автоматом из упражнения 3.5.

3.7. Число с фиксированной точкой определяется как возможный знак, за которым следует строка, состоящая из нуля или большего числа цифр, с последующей точкой и следующей за нею строкой из нуля или более цифр; содержит она по крайней мере одну цифру (до или после точки). Напишите процедуру распознавания такого числа.

3.8. Товарный поезд может состоять из одного или двух локомотивов, одной или более платформ и следующего за ними вагона с охраной. Напишите процедуру распознавания товарного поезда.

3.9. Предложите структуры данных, применимые для реализации таблицы констант.

3.10. Предложите вместо взаимно рекурсивных процедур другой вариант для обращения с двумя видами лексического анализа, требуемого при использовании Алгола 68 (т. е. внутри и вне форматов).

Глава 4

КОНТЕКСТНО-СВОБОДНЫЕ ГРАММАТИКИ И СИНТАКСИЧЕСКИЙ АНАЛИЗ СВЕРХУ ВНИЗ

В предыдущей главе мы рассмотрели проблему распознавания языков типа 3 и показали, что конечный автомат можно применять для распознавания любого языка этого типа. Далее мы определили, что во избежание возврата всегда можно воспользоваться детерминированным автоматом. Грамматики типа 3 удобны для генерирования символов, которые создаются во время лексического анализа, но они не очень подходят для описания способов объединения этих символов в предложения типичных языков программирования. Например, как уже отмечалось в гл. 2, согласование скобок нельзя специфицировать с помощью грамматики типа 3. Контекстно-свободные грамматики, хотя и не могут специфицировать все свойства типичных языков программирования, являются более универсальными и поэтому более пригодными в качестве основы для фазы синтаксического анализа (разбора) компиляции.

Здесь мы изучим свойства контекстно-свободных грамматик и их связь с теорией автоматов, а также приведем методы синтаксического анализа сверху вниз. Методы синтаксического анализа снизу вверх описываются в гл. 5.

4.1. КОНТЕКСТНО-СВОБОДНЫЕ ГРАММАТИКИ

Контекстно-свободные грамматики традиционно служат основой для синтаксического анализа компиляции. Когда какой-либо язык программирования нельзя генерировать с помощью контекстно-свободной грамматики, всегда можно найти такую контекстно-свободную грамматику, которая генерирует супермножество языка, т. е. язык, включающий в себя заданный язык программирования. Применение этой грамматики для синтаксического анализа означает, что ряд ограничений в реальном языке (например, обязательное объявление всех идентификаторов в программе) не будет проверяться анализатором. Однако в компиляторе нетрудно предусмотреть другие действия по выполнению необходимых проверок в исходной программе (например, за счет применения таблицы символов).

Из определения контекстно-свободной грамматики, приведенного в гл. 2, ясно, что класс контекстно-свободных грамматик более мощный (т. е. может генерировать больше языков), чем класс регулярных грамматик, но менее мощный, чем класс контекстно-зависимых грамматик. Язык

$$\{a^n b^n \mid n \geq 0\}$$

является контекстно-свободным, но не регулярным. Он генерируется грамматикой с порождающими правилами

$$S \to aSb$$
$$S \to \varepsilon$$

С другой стороны, язык

$$\{a^n b^n c^n \mid n > 0\}$$

является контекстно-зависимым, а не контекстно-свободным.

Контекстно-свободные грамматики имеют ряд характеристик, для которых справедливы следующие положения. Доказательства большинства из них можно найти в [27].

1. *Каноническая форма*

а) Каждая контекстно-свободная грамматика эквивалентна (т. е. генерирует тот же язык) грамматике в *нормальной форме Хомского*, т. е. со всеми порождающими правилами вида

$$A \to BC$$
$$A \to a$$

при обычных условиях, касающихся терминалов и нетерминалов.

б) Каждая контекстно-свободная грамматика эквивалентна грамматике в *нормальной форме Грейбаха*, т. е. со всеми порождающими правилами вида

$$A \to b\alpha$$

где α — строка нетерминалов (возможно, пустая).

2. Самовложение

Самовложение определяется следующим образом. Если в грамматике G есть нетерминал A, для которого

$$A \overset{*}{\underset{G}{\Rightarrow}} \alpha_1 A \alpha_2$$

(здесь α_1 и α_2 являются непустыми строками терминалов и/или нетерминалов), то о такой грамматике говорят, что она содержит *самовложение*. Например, две приведенные ниже грамматики содержат самовложение:

1) $G_1 = (\{S\}, \{a, b\}, P, S)$,

где элементы P:

$$S \to aSb$$
$$S \to \varepsilon$$

2) $G_2 = (\{S, A\}, \{\textbf{begin}, \textbf{end}, [,]\}, P, S)$,

где элементы P:

$$S \to \textbf{begin}\ A\ \textbf{end}$$
$$S \to \varepsilon$$
$$A \to [S]$$

В последнем случае об A и S говорят, что они проявляют свойство самовложения. Теоретически любая контекстно-свободная грамматика, не содержащая самовложения, эквивалентна регулярной грамматике или (иначе) генерирует регулярный язык. Дело также в том,

что регулярная грамматика не может содержать самовложения. Именно самовложение позволяет эффективно различать контекстно-свободные (нерегулярные) и регулярные языки. Как видно из второго примера, согласование скобок и т. п. требует самовложения, поэтому его нельзя специфицировать посредством регулярной грамматики.

3. *Лемма подкачки*

Для любого контекстно-свободного языка L существует такая константа k, что любое предложение языка z, длина которого превышает k, можно записать в виде

$$z = uvwxy, \text{ где } |vwx| \leqslant k,$$

v и x не являются пустой строкой, а строка

$$uv^i wx^i y \quad (\text{для } i > 0)$$

находится в L.

С помощью этой леммы можно продемонстрировать, что определенные языки не являются контекстно-свободными. Рассмотрим язык

$$\{aa \mid a \in \{0,1\}^*\}$$

($\{0,1\}^*$ есть замыкание $\{0,1\}$, как определено в разд. 2.2). Из леммы следует, что любую строку z длиной $> k$ можно записать как в виде $uvwxy$, где $|vwx| \leqslant k$, $|vx| \neq 0$ и $uv^i wx^i y (i \geqslant 0)$ находится в языке.

Пусть α состоит из k нулей и последующих k единиц, так что z имеет вид

$$000\cdots01111\cdots1000\cdots01111\cdots1$$

Эту строку можно записать следующим образом:

$$uvwxy,$$

но так как $|vwx| \leqslant k$, то или

1) v и x находятся либо в первой половине z, либо во второй, или

2) vx содержит единицы из первой половины строки и нули из второй половины строки.

В первом случае uwy образуется путем исключения нулей и единиц из первой или второй половины z (но не из обеих). Таким образом, uwy не находится в языке.

Во втором случае uwy образуется из z путем исключения единиц из ее первой половины и нулей из второй. Полученная в результате строка опять не будет находиться в языке.

Таким образом, мы показали, что данная строка не может быть выражена в виде $uvwxy$, чтобы uwy тоже находилась в языке. Это противоречит лемме подкачки, и поэтому данный язык не относится к контекстно-свободным.

Аналогично можно показать, что языки

$$\{a^n b^n c^n \mid n \geqslant 0\}$$
$$\{a^i \mid i \text{ есть простое число}\}$$

не являются контекстно-свободными.

С точки зрения разбора важно знать, какой тип автомата в состоянии распознавать контекстно-свободный язык. Им может быть *автомат магазинного типа*, эквивалентный конечному автомату, к которому до-

бавлена память магазинного типа или стек. В функции автомата магазинного типа входит

а) чтение входного символа, замещение верхнего символа стека строкой символов (возможно, пустой) и изменение состояния или

б) все то же самое, но без чтения входного символа.

Автомат магазинного типа можно представить кратным

$$(K,\Sigma,\Gamma,\delta,q_0,Z_0),$$

где K — конечное множество состояний, Σ — входной алфавит, Γ — алфавит магазинный, δ — множество переходов, q_0 — начальное состояние, Z_0 — символ магазина, который первоначально находится в стеке.

Рассмотрим, например, автомат магазинного типа M, определенный следующим образом:

$$K=\{A\},$$

$$\Sigma=\{'(',')'\},$$
$$\Gamma=\{O,I\},$$
$$q_0=\{A\},$$
$$Z_0=I.$$

δ задается как

$$\delta(A,I,'(')=(A,IO)$$

(что означает: в состоянии A с I в вершине стека при чтении '(' перейти к состоянию A и заменить I на IO).

$$\delta(A,O,'(')=(A,OO),$$
$$\delta(A,O,')')=(A,\varepsilon),$$
$$\delta(A,I,\varepsilon)=(A,\varepsilon).$$

Автомат M распознает согласуемые пары скобок. Открывающие скобки (представляемые как O) помещаются в стек и удаляются оттуда, когда встречается соответствующая закрывающая скобка. Строка скобок принимается, если после считывания всей строки стек остается пустым. Это — обычный способ принятия строк автоматами магазинного типа, хотя можно также определить автоматы магазинного типа, которые принимают строки по конечному состоянию. Эти два типа эквивалентны.

Описанный выше автомат магазинного типа является детерминированным, т. е. для каждого допустимого входного символа имеется однозначный переход. Что же касается конечных автоматов, то мы можем также определять недетерминированные автоматы магазинного типа, содержащие *множество* переходов для заданного входа, состояния и содержания стека.

Рассмотрим язык

$$\{\alpha\alpha^r \mid \alpha \text{ в } \{0,1\}^*\}.$$

Здесь α^r обозначает строку, обратную α.

Этот язык принимается следующим недетерминированным автоматом магазинного типа

$$M=(\{A,B\},\{0,1\},\{I,0,1\},\delta,A,I),$$

где переходы заданы посредством

$$\delta(A,1,0) = \{(A,10)\},$$
$$\delta(A,1,1) = \{(A,11)\},$$
$$\delta(A,0,0) = \{(A,00),(B,\varepsilon)\},$$
$$\delta(A,1,1) = \{(A,11),(B,\varepsilon)\},$$
$$\delta(A,0,1) = \{(A,01)\},$$
$$\delta(A,1,0) = \{(A,10)\},$$
$$\delta(B,0,0) = \{(B,\varepsilon)\},$$
$$\delta(B,1,1) = \{(B,\varepsilon)\},$$
$$\delta(B,1,\varepsilon) = \{(B,\varepsilon)\},$$
$$\delta(A,1,\varepsilon) = \{(A,\varepsilon)\}.$$

Состояние A соответствует прочтению менее половины строки, а состояние B — прочтению более половины строки. При чтении первой половины строки символы помещаются в стек (состояние A), а при чтении второй — каждый символ сверяется с верхним символом в стеке, и из стека удаляется один символ. Задача заключается в том, чтобы установить, когда достигается середина строки. При этом должны считываться две последовательные единицы или два последовательных нуля. Это условие является необходимым, но недостаточным. Возьмем, например, строку

0101101101101010

Можно подумать, что две единицы, занимающие позиции 4 и 5, отмечают середину строки. Однако, не заглянув вперед (в целом) на какое-либо произвольное расстояние, нельзя с уверенностью сказать, середина ли это. Отсюда — попеременные переходы в состоянии A, когда входной символ и вершина стека являются идентичными. Это делает автомат магазинного типа недетерминированным, и трудно представить себе иную ситуацию. Действительно, как известно из теории, существует недетерминированный автомат магазинного типа, принимающий контекстно-свободные языки, которые не принимаются никакими детерминированными автоматами магазинного типа. Следовательно, мы всегда можем найти детерминированный автомат, принимающий регулярный язык. Для контекстно-свободных же языков это положение не выдерживается. Если вспомнить, что детерминированные конечные автоматы получают из недетерминированных путем соединения состояний, где два или более переходов приводят к различным состояниям, вряд ли покажется странным, что такой же метод не срабатывает в отношении автоматов магазинного типа, когда альтернативные переходы могут по-разному влиять на стек.

При разборе происходит эффективное моделирование соответствующего автомата магазинного типа. А из изложенного выше вытекает, что некоторые контекстно-свободные языки не могут анализироваться детерминированным образом (т. е. без возврата). Насколько это важно, с точки зрения создателя компилятора, зависит от того, можно ли типичные языки программирования анализировать детерминированно или нет. Язык, который допускает детерминированный анализ, мы называем *детерминированным*; оказывается, что большинство языков программирования являются детерминированными или по крайней мере почти таковыми.

Между контекстно-свободными грамматиками и автоматами магазинного типа существует полное соответствие, и детерминированность автомата может зависеть от того, какая грамматика используется для генерирования языка. Мы говорим, что метод разбора является детерминированным (для конкретной грамматики), если при разборе данной грамматики не требуется делать возврат. Некоторые языки можно разбирать детерминированно с помощью только одного из методов грамматического разбора. В частности, ряд языков можно разбирать детерминированно снизу вверх, но не сверху вниз. В этой книге мы будем иметь дело почти исключительно с детерминированными методами разбора. Недетерминированные методы могут применяться к таким строчно-ориентированным языкам, как Бейсик или Фортран. Для языков же крайне рекурсивных (Алгол 68, Паскаль), где компилятор может быть вынужден возвращаться назад не только в текущей строке, но и в большой части программы, издержки просто неприемлемы. Другой недостаток возврата заключается в том, что он может вызвать отмену действий компилятора, которые осуществляются параллельно с синтаксическим анализом.

4.2. МЕТОД РЕКУРСИВНОГО СПУСКА

Метод рекурсивного спуска — хорошо известный и легко реализуемый детерминированный метод разбора сверху вниз. С его помощью на основании соответствующей грамматики можно написать синтаксический анализатор, причем почти так же быстро, как мы можем вообще писать.

Рассмотрим, например, язык, генерируемый грамматикой со следующими порождающими правилами:

PROGRAM→begin DECLIST comma STATELIST end
DECLIST→d semi DECLIST
DECLIST→d
STATELIST→s semi STATELIST
STATELIST→s

Сначала видоизменим грамматику:

PROGRAM→begin DECLIST comma STATELIST end
DECLIST→*d X*
X→semi DECLIST
X→ε
STATELIST→s Y
Y→semi STATELIST
Y→ε

Теперь напишем процедуру распознавания каждого нетерминала. Начав с символа (*PROGRAM*), получим

```
proc PROGRAM = void:
begin if symbol ≠ begin then error fi;
   symbol := lexical;
   DECLIST;
   if symbol ≠ comma then error fi;
   symbol := lexical;

   STATELIST;
   if symbol ≠ end then error fi
end;
```

```
proc DECLIST = void:
begin if symbol ≠ d then error fi;
  symbol := lexical;
  X
end;

proc X = void:
if symbol = semi then
symbol := lexical; DECLIST
elif symbol = comma
then skip
else error
fi;

proc STATELIST = void:
begin if symbol ≠ s then error fi;
  symbol := lexical;
  Y
end;

proc Y = void:
if symbol = semi then
symbol := lexical; STATELIST
elif symbol = end then skip
else error
fi;
```

В этих процедурах сделан ряд допущений:

1. *error* — процедура обращения с синтаксическими ошибками; детали нас здесь не интересуют;

2. *lexical* — процедура, которая читает литеры исходного текста и выдает следующий символ. Если лексический анализ выполняется предыдущим проходом, *lexical* просто читает следующий символ;

3. *semi, comma* и т. д. — идентификаторы, значениями которых являются представления соответствующих терминалов во время компиляции;

4. прежде чем вызывается *PROGRAM*, происходит следующее присваивание:

$$symbol := lexical.$$

Для разбора предложения языка может потребоваться много рекурсивных вызовов процедур, соответствующих нетерминалам в грамматике. Если мы решим представлять грамматику несколько иным путем, эту рекурсию можно заменить итерацией:

$$PROGRAM \rightarrow begin\ DECLIST\ comma\ STATELIST\ end$$
$$DECLIST \rightarrow d\ (semi\ d)^*$$
$$STATELIST \rightarrow s\ (semi\ s)^*$$

где $(x)^*$ означает нуль или более реализаций x. Тогда процедуры для *DECLIST* и *STATELIST* становятся такими:

```
proc DECLIST = void:
begin if symbol ≠ d then error fi;
  symbol := lexical;
  while symbol = semi
  do symbol := lexical;
```

```
        if symbol ≠ d then error fi;
        symbol := lexical
    od
end;

proc STATELIST = void:
begin if symbol ≠ s then error fi;
symbol := lexical;
while symbol = semi
do symbol := lexical;
        if symbol ≠ s then error fi;
        symbol := lexical
    od
end
```

Замена рекурсии итерацией, возможно, делает анализатор более эффективным, а также (не бесспорно) более удобочитаемым.

Преимущества написания рекурсивного нисходящего анализатора очевидны. Основные из них — это скорость написания анализатора на основании соответствующей грамматики. Другое преимущество заключается в соответствии между грамматикой и анализатором, благодаря которому увеличивается вероятность того, что анализатор окажется правильным, или, по крайней мере, того, что ошибки будут носить простой характер. Недостатки этого метода, хотя и менее очевидны, не менее реальны. Из-за большого числа вызовов процедур во время синтаксического анализа анализатор становится относительно медленным. Кроме того, он может быть довольно большим по сравнению с анализаторами, основанными на табличных методах разбора, которые описываются ниже. Несмотря на то что данный метод способствует включению в анализатор действия по генерированию кода, это неизбежно приводит к смешиванию различных фаз компиляции. Последнее снижает надежность компиляторов или усложняет обращение с ними и как бы привносит в анализатор зависимость от машины. Рекурсивный метод разбора сверху вниз родствен другому хорошо известному методу разбора сверху вниз, применение которого позволяет избежать большинства указанных недостатков. Этот метод рассматривается в следующем разделе.

4.3. LL(1)-ГРАММАТИКИ

LL(1)-грамматика — это грамматика такого типа, на основании которой можно получить детерминированный синтаксический анализатор, работающий по принципу сверху вниз. Прежде чем определить LL(1)-грамматику более точно, введем понятие *s*-грамматики.

Определение. *s*-грамматика представляет собой грамматику, в которой:

1) правая часть каждого порождающего правила начинается с терминала;

2) в тех случаях, когда в левой части более чем одного порождающего правила появляется нетерминал, соответствующие правые части начинаются с разных терминалов.

Первое условие аналогично утверждению, что грамматика находится в нормальной форме Грейбаха, только за терминалом в начале каждой правой части правила могут следовать нетерминалы *и/или* терминалы.

Второе условие помогает нам написать детерминированный нисходящий анализатор, так как при выводе предложения языка всегда можно сделать выбор между альтернативными порождающими правилами для самого левого нетерминала в сентенциальной форме, предварительно исследовав один следующий символ.

Грамматика с порождающими правилами

$$\begin{array}{ll} S \to pX & X \to x \\ S \to qY & Y \to aYd \\ X \to aXb & Y \to y \end{array}$$

представляет собой s-грамматику, тогда как следующая грамматика, которая генерирует тот же язык, не является ею:

$$\begin{array}{ll} S \to R & X \to aXb \\ S \to T & X \to x \\ R \to pX & Y \to aYd \\ T \to qY & Y \to y \end{array}$$

поскольку правые части двух правил не начинаются с терминалов.

Определить, используется ли в качестве заданной грамматики s-грамматика, очень легко, и в некоторых случаях грамматику, которая не является s-грамматикой, можно преобразовать в нее, не затрагивая при этом генерируемый язык.

Рассмотрим проблему разбора строки

paaaxbbb

с помощью s-грамматики. Начав с символа S, попытаемся генерировать строку. Применим левосторонний вывод. Результат приводится ниже.

Исходная строка	Вывод
paaaxbbb	S
paaaxbbb	*pX*
paaaxbbb	*paXb*
paaaxbbb	*paaXbb*
paaaxbbb	*paaaXbbb*
paaaxbbb	*paaaxbbb*

При выводе начальные терминалы в сентенциальной форме сверяются с символами в исходной строке. Стрелка показывает, до какой точки на каждом этапе был считан исходный текст и проверены начальные терминалы. Там, где допускается замена самых левых терминалов в сентенциальной форме с помощью более чем одного порождающего правила, всегда можно выбрать соответствующее порождающее правило, исследовав следующий символ во входной строке. Это связано с тем, что, поскольку мы имеем дело с s-грамматикой, правые части альтернативных порождающих правил будут начинаться с различных терминалов.

Таким образом, всегда можно написать детерминированный анализатор, осуществляющий разбор сверху вниз, для языка, генерируемого s-грамматикой. LL(1)-грамматика является обобщением s-грамматики, и, как мы увидим далее, принцип ее обобщения все еще позволяет нам строить нисходящие детерминированные анализаторы.

Две буквы L в LL(1) означают, что строки разбираются слева направо и используются самые левые выводы, а цифра 1 — что варианты порождающих правил выбираются с помощью одного предварительно просмотренного символа. Эта терминология была введена Кнутом [36]. Свойства же LL(1)-грамматик изучались также Фостером [19]. Следует отметить, что, хотя правая часть порождающего правила и не начинается с терминала, заданный вариант какого-либо нетерминального символа может дать начало только тем строкам, которые начинаются с одного из конкретного множества терминалов. Например, на основании порождающих правил

$$\begin{aligned} &S \to RY \\ &S \to TZ \\ &R \to paXb \\ &T \to qaYd \end{aligned}$$

можно вывести, что порождающее правило $S \to RY$ желательно применять только в разборе сверху вниз (допуская, что для R нет других порождающих правил), когда предварительно просматриваемым символом является p. Аналогично порождающее правило $S \to TZ$ рекомендуется в тех случаях, когда таким символом окажется q.

Это приводит нас к идее множеств символов-предшественников, определяемых как

$$a \in S(A) \Leftrightarrow A \Rightarrow a\alpha,$$

где A — нетерминальный символ, α — строка терминалов и/или нетерминалов, а $S(A)$ обозначает множество символов-предшественников A. В грамматике с порождающими правилами

$$\begin{aligned} &P \to Ac \qquad A \to aA \\ &P \to Bd \qquad B \to b \\ &A \to a \qquad B \to bB \end{aligned}$$

a и b — символы-предшественники для P. Определим также множество символов-предшественников для строки терминалов и/или нетерминалов:

$$a \in S(\alpha) \Leftrightarrow \alpha \Rightarrow a\beta$$

Здесь α и β есть строки терминалов и/или нетерминалов (β может быть пустой строкой).

Необходимым условием для того, чтобы грамматика обладала признаком LL(1), является непересекаемость множеств символов-предшественников для альтернативных правых сторон порождающих правил.

Следует проявлять осторожность в тех случаях, когда нетерминал в начале правой части может генерировать пустую строку. Например, в

$$\begin{aligned} &P \to AB \qquad A \to e \\ &P \to BG \qquad B \to c \\ &A \to aA \qquad B \to bB \end{aligned}$$

имеем

$$S(AB)=\{a,b,c\}$$

поэтому A может генерировать пустую строку и

$$S(BG)=\{b,\ c\}$$

так что грамматика не будет LL(1).

Рассмотрим также грамматику с порождающими правилами

$T\to AB$
$A\to PQ$
$A\to BC$
$P\to pP$
$P\to\varepsilon$
$Q\to qQ$
$Q\to\varepsilon$
$B\to bB$
$B\to e$
$C\to cC$
$C\to f$

которая дает

$$S(PQ)=\{p,q\}$$

и

$$S(BC)=\{b,e\}$$

Однако так как PQ может генерировать пустую строку, следующим просматриваемым символом при применении порождающего правила $A\to PQ$ может быть b или e (вероятные символы, следующие за A), и одного следующего просматриваемого символа недостаточно, чтобы различить две альтернативные правые части для A. Последнее связано с тем, что b и e являются также символами-предшественниками для BC.

Поэтому мы вводим *направляющие символы*, которые по Гриффитсу [22] определяются так: если A — нетерминал, то его направляющими символами будут

$S(A)+$ (все символы, следующие за A, если A может генерировать пустую строку).

В более общем случае для заданного варианта α нетерминала P $(P\to\alpha)$ имеем

$$DS\ (P,\alpha)=\{a\mid a\in S(\alpha)\ \text{или}\ (\alpha\overset{*}{\Rightarrow}\varepsilon\ \text{и}\ a\in F(P))\}$$

где $F(P)$ есть множество символов, которые могут следовать за P. Так, в приведенном ранее примере направляющие символы — это символы

$$DS\ (A,PQ)=\{p,q,b,e\}$$
$$DS\ (A,BC)=\{b,e\}$$

Поскольку указанные множества пересекаются, данная грамматика не может служить основой для детерминированного нисходящего анализатора, использующего один предварительно просматриваемый символ для различения альтернативных правых частей.

Теперь мы готовы определить LL(1)-грамматику. Грамматику называют LL(1)-грамматикой, если для каждого нетерминала, появляющегося в левой части более одного порождающего правила, множества направляющих символов, соответствующих правым частям альтернативных порождающих правил, — непересекающиеся. Все LL(1)-грамматики можно разбирать детерминированно сверху вниз.

Существует алгоритм, который позволяет выяснить, представляет ли собой заданная грамматика LL(1)-грамматику. Опишем этот алгоритм.

Прежде всего нужно установить, какие нетерминалы могут генерировать пустую строку. Для этого создадим одномерный массив, где каждому нетерминалу соответствует один элемент. Любой элемент массива может принимать одно из трех значений: *YES, NO* или *UNDECIDED*. Вначале все элементы имеют значение *UNDECIDED*. Мы просматриваем грамматику столько раз, сколько требуется для того, чтобы каждый элемент принял значение *YES* или *NO*.

При первом просмотре исключаются все порождающие правила, содержащие терминалы. Если это приводит к исключению всех порождающих правил для какого-либо нетерминала, соответствующему элементу массива присваивается значение *NO*. Затем для каждого порождающего правила с ε в правой части тому элементу массива, который соответствует нетерминалу в левой части, присваивается значение *YES*, и все порождающие правила для этого нетерминала исключаются из грамматики.

Если требуются дополнительные просмотры (т.е. значения некоторых элементов массива имеют все еще значение *UNDECIDED*), выполняются следующие действия:

1. Каждое порождающее правило, имеющее такой символ в правой части, который не может генерировать пустую строку (о чем свидетельствуют значения соответствующего элемента массива), исключается из грамматики. В том случае, когда для нетерминала в левой части исключенного правила не существует других порождающих правил, значение элемента массива, соответствующего этому нетерминалу, устанавливается на *NO*.

2. Каждый нетерминал в правой части порождающего правила, который может генерировать пустую строку, стирается из правила. В том случае, когда правая часть правила становится пустой, элементу массива, соответствующему нетерминалу в левой части, присваивается значение *YES*, и все порождающие правила для этого нетерминала исключаются из грамматики.

Этот процесс продолжается до тех пор, пока за полный просмотр грамматики не изменится ни одно из значений элементов массива. Если допустить, что вначале грамматика была «чистой» (т. е. все нетерминалы могли генерировать конечные или пустые строки), то теперь все значения элементов массива будут установлены на *YES* или *NO*. Рассмотрим этот процесс на примере грамматики

1. $A \to XYZ$	7. $Q \to aa$
2. $X \to PQ$	8. $Q \to \varepsilon$
3. $Y \to RS$	9. $S \to cc$
4. $R \to TU$	10. $T \to dd$
5. $P \to \varepsilon$	11. $U \to ee$
6. $P \to a$	12. $Z \to \varepsilon$

После первого прохода массив будет таким, как показано в табл 4.1, а грамматика сведется к следующей:

1. $A \rightarrow XYZ$
2. $X \rightarrow PQ$
3. $Y \rightarrow RS$
4. $R \rightarrow TU$

Таблица 4.1

A	X	Y	R	P	Q	S	T	U	Z
U	U	U	U	Y	Y	N	N	N	Y

U — *UNDECIDED* (НЕРЕШЕННЫЙ)
Y — *YES* (ДА)
N — *NO* (НЕТ)

После второго прохода массив станет таким, как показано в табл. 4.2, а грамматика примет вид

1. $A \rightarrow XY$

Таблица 4.2

A	X	Y	R	P	Q	S	T	U	Z
U	Y	N	N	Y	Y	N	N	N	Y

Третий проход завершает заполнение массива (табл. 4.3).

Таблица 4.3

A	X	Y	R	P	Q	S	T	U	Z
N	Y	N	N	Y	Y	N	N	N	Y

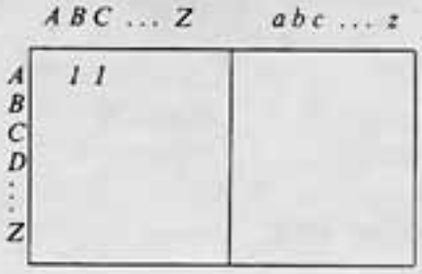
Далее формируется матрица, показывающая всех *непосредственных предшественников* каждого нетерминала. Этот термин используется для обозначения тех символов, которые из одного порождающего правила уже видны как предшественники. Например, на основании правил

$$P \rightarrow QR$$
$$Q \rightarrow qR$$

можно заключить, что Q есть непосредственный предшественник P, а q — непосредственный предшественник Q. В матрице предшественников для каждого нетерминала отводится строка, а для каждого терминала и нетерминала — столбец. Если нетерминал A, например, имеет в качестве непосредственных предшественников B и C, в A-ю строку в B-м и C-м столбцах помещаются единицы (табл. 4.4).

Там, где правая часть правила начинается с нетерминала, необходимо проверить, может ли данный нетерминал генерировать пустую строку, для чего используется массив пустой строки. Если такая ге-

Таблица 4.4.



нерация возможна, символ, следующий за нетерминалом (при их наличии), является непосредственным предшественником нетерминала в левой части правила и т. д.

Как только непосредственные предшественники будут введены в матрицу, мы сможем делать следующие заключения. Например, из порождающих правил

$$P \to QR$$
$$Q \to qV$$

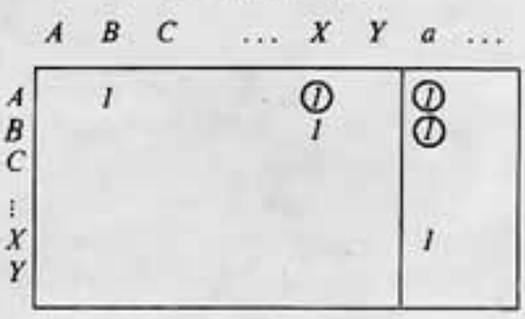
можно заключить, что q есть символ (не непосредственного) предшественника P. Или же, как вытекает из матрицы непосредственных предшественников, единица в P-й строке Q-го столбца и единица в Q-й строке q-го столбца свидетельствуют о том, что если мы хотим сформировать полную матрицу предшественников (а не только непосредственных), нам нужно поместить единицу в P-ю строку q-го столбца. В обобщенном варианте, когда в (i, j)-й позиции u в (j,k)-й позиции стоят единицы, нам следует поставить единицу в (i,k)-ю позицию. Пусть, однако, и в (k,l)-й позиции также стоит единица. Тогда этот процесс рекомендуется выполнить вновь, чтобы поставить единицу в (i,l)-ю позицию, и повторять его до тех пор, пока не будет таких случаев, когда в (i,j)-й позиции и в (j,k)-й позиции появляются единицы, а в (i,k)-й позиции — нет.

Приведенный выше алгоритм иллюстрируется множеством порождающих правил:

$$A \to BC$$
$$B \to XY$$
$$X \to aa$$

из которых легко ввести три единицы в матрицу непосредственных предшественников и путем дедукции еще три единицы — в полную матрицу предшествования в соответствии с тем, что X является не непосредственным предшественником A, а a — не непосредственным предшественником B, а также A (табл. 4.5). В таблице не непосредственные

Таблица 4.5.



предшественники заключены в кружки.

Этот процесс известен как нахождение *транзитивного замыкания* матрицы, и у него есть аналог в теории графов. Например, на основании рис. 4.1 мы можем построить так называемую матрицу смежности, в которой содержатся единицы, указывающие на соседние вершины. Например, вершина A является соседней с C, так что в позиции (A,C) будет проставлена единица, и т. д. (Считается, что A — вершина, соседняя самой себе.) С помощью такой матрицы можно выполнить операцию транзитивного замыкания и получить матрицу достижимости.

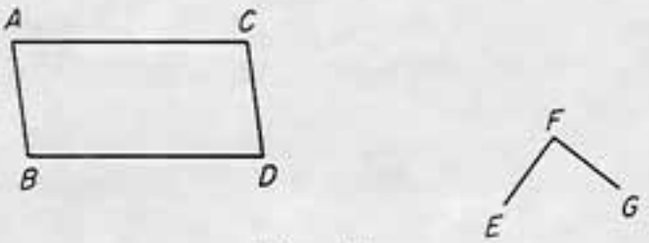


Рис. 4.1

В матрице достижимости единицы соответствуют вершинам, между которыми есть соединительные пути (табл. 4.6. и 4.7).

Таблица 4.6

Матрица смежности

	A	B	C	D	E	F	G
A	1	1	1				
B	1	1		1			
C	1		1	1			
D		1	1	1			
E					1	1	
F					1	1	1
G						1	1

Таблица 4.7

Матрица достижимости

	A	B	C	D	E	F	G
A	1	1	1	1			
B	1	1	1	1			
C	1	1	1	1			
D	1	1	1	1			
E					1	1	1
F					1	1	1
G					1	1	1

Читатель, вероятно, знаком с алгоритмом выполнения транзитивного замыкания по алгоритму Уоршалла [56].

Приведем версию этого алгоритма на Алголе 68. Вначале массив A представляет собой матрицу смежности (или содержит непосредственных предшественников), а в конце трансформируется в матрицу достижимости (или включает всех предшественников).

Считается, что индексы в A начинаются с 1.

```
proc Warshall = (ref [ , ] int A) void :
begin int upb1 = 1 upb A, upb2 = 2upb A :
  for i to upb1
  do for j to upb1
    do if A[i,j] = 1 then
      for k to upb2
      do if A[j,k] = 1 then
        A[i,k] := 1 fi
      od fi
    od
  od
end
```

Может показаться, что данной процедуре потребуется внешний цикл, чтобы повторять процесс до тех пор, пока в A не произойдут изменения. Однако по теореме Уоршалла [56] это не является необходимым. Алгоритму Уоршалла для построения $n \times n$ матрицы нужно время, пропорциональное n^3. Тем не менее с помощью работы Стразена [53] можно доказать, что существует алгоритм, которому нужно время, пропорциональное $n^{2 \cdot 81}$. Можно также воспользоваться относительно неплотным характером матрицы непосредственных предшественников и получить более эффективный алгоритм [21].

Если бы не проблема нетерминалов, генерирующих пустую строку, для LL(1)-проверки, потребовалось бы изучение только матрицы предшествования. Но мы уже знаем, что при вычислении направляющих символов иногда приходится рассматривать символы, которые по праву могут следовать за определенным нетерминалом. Поэтому мы строим матрицу следования из порождающих правил грамматики. Например, как вытекает из правила

$$S \rightarrow ABC$$

B может следовать за A, и элемент (A, B) в матрице следования имеет значение *1*. Аналогичным образом C может следовать за B, и элемент (B, C) устанавливается на *1*. Если B генерирует пустой символ, то естественно заключить, что C может следовать за A, и элемент (A, C) устанавливается на *1*.

Менее явно правила

$$P \rightarrow QR$$
$$Q \rightarrow VU$$

позволяют сделать вывод о том, что R следует за U или, в более общем виде (на основании второго правила), что «любой символ, следующий за Q, следует и за U».

Заметим также, что если, например, B следует за A и b есть предшественник B, то b следует и за A. Таким образом, матрицу предшествования можно использовать для того, чтобы вывести больше символов-следователей и соответственно нарастить матрицу следования.

На основании массива пустых строк, матрицы предшествования и матрицы следования можно проверить признак LL(1). Там, где в левой части более чем одного правила появляется нетерминал, необходимо вычислять направляющие символы различных альтернативных правых частей. Если для каких-либо из этих нетерминалов различные множества направляющих символов не являются непересекающимися, грамматика окажется не LL(1). В противном случае она будет LL(1).

В результате преобразования, выполненного в начале разд. 4.2, грамматика, которая не являлась LL(1)-грамматикой, была преобразована в нее (см. варианты для *DECLIST* и *STATELIST*), а именно:

PROGRAM→*begin DECLIST comma STATELIST end*
DECLIST→*d X*
X→*semi DECLIST*
X→ε
STATELIST→*s Y*
Y→*semi STATELIST*
Y→ε

С целью проверки признака LL(1) этой грамматики образуем различные матрицы (табл. 4.8 — 4.10). Необходимо только рассмотреть варианты для X и Y.

Таблица 4.8

Массив пустых строк

PROGRAM	*DECLIST*	*STATELIST*	*X*	*Y*
N	N	N	Y	Y

Полная матрица предшествования

Таблица 4.9

	PROGRAM	*DECLIST*	*STATELIST*	*X*	*Y*	**begin**	*d*	*s*	*comma*	*semi*	**end**
PROGRAM											
DECLIST											
STATELIST											
X											
Y											

Матрица следования

Таблица 4.10

	PROGRAM	*DECLIST*	*STATELIST*	*X*	*Y*	**begin**	*d*	*s*	*comma*	*semi*	**end**
PROGRAM											
DECLIST											
STATELIST											
X											
Y											

Первый вариант для X имеет множество направляющих символов {*semi*}. Второй вариант — пустая строка, поэтому нам нужно выяснить, что может следовать за X. Единственным символом-следователем служит *comma*, так что множеством направляющих символов будет {*comma*}. Множества направляющих символов являются непересекающимися, следовательно, нарушения условия LL(1) в отношении порождающих правил для нетерминала X нет. Аналогичным образом можно показать, что множествами направляющих символов для Y будут {*semi*} и {*end*}, и поэтому мы имеем дело с LL(1)-грамматикой.

4.4. LL(1)-ЯЗЫКИ

Мы уже убедились в том, что существует алгоритм для определения признака LL(1)-грамматики. В этой связи возникают два вопроса:

1. Все ли языки обладают LL(1)-грамматикой?
2. Если нет, то существует ли алгоритм для определения свойства LL(1) языка (т. е. можно ли его генерировать с помощью LL(1)-грамматики) или нет?

Ответ на первый вопрос отрицательный. Например, язык

$$\{\alpha\alpha^r | \alpha \text{ в } \{0, 1\}^*\}$$

генерированный грамматикой с правилами

$$S \to 0S0$$
$$S \to 1S1$$
$$S \to \varepsilon$$

не имеет свойства LL(1). Как было показано в разд. 4.1, этот генерированный язык не принимается ни одним детерминированным автоматом магазинного типа (что фактически не было доказано), поэтому не генерируется никакой LL(1)-грамматикой. Таким образом, мы видим, что грамматика, которая не обладает признаком LL(1), может генерировать язык LL(1), а может и не генерировать его. Иными словами, грамматику, не являющуюся LL(1), либо можно преобразовать в LL(1)-грамматику, генерирующую тот же язык, либо нельзя. Здесь уместно рассмотреть второй вопрос. Существует ли алгоритм для определения свойства LL(1)-языка или нет? Ответ на этот вопрос также отрицательный, и о проблеме говорят, что она неразрешима. Из теории известно, что такого алгоритма не существует (по крайней мере, алгоритма, который с гарантией сработал бы в любом случае). Все попытки получить подобный алгоритм могут привести к бесконечному зацикливанию в определенных ситуациях.

Насколько важен этот результат на практике? Оказывается, что «очевидной» грамматикой для большинства языков программирования является не LL(1)-грамматика. Однако обычно очень большое число контекстно-свободных средств языка программирования можно представить с помощью LL(1)-грамматики. Проблема заключается в том, чтобы, имея грамматику, которая не обладает признаком LL(1), найти эквивалентную ей LL(1)-грамматику. Так как алгоритм, позволяющий определить, существует ли такая LL(1)-грамматика или нет, отсутствует, значит, отсутствует и алгоритм, который мог бы определить, существуют ли соответствующие преобразования, и выполнить их. В некоторых случаях, однако, эти преобразования легко наметить. Например, преобразования для грамматики, приведенной в начале разд. 4.2, весьма просты. Многие разработчики компиляторов, обладающие достаточным опытом, не испытывают затруднений в преобразовании грамматик вручную с целью устранения их свойств, отличных от LL(1). Тем не менее с ручным преобразованием связаны серьезные опасения. Основное из них заключается в том, что человек, выполняющий это преобразование с самыми лучшими намерениями, может случайно изменить язык, генерируемый данной грамматикой. Если в наши задачи входит создание надежных компиляторов, то нам по мере возможности желательно избегать преобразований вручную.

Но если не существует полностью универсального автоматического процесса преобразования грамматик в LL(1)-форму, то что же остается нам делать? Отсутствие общего решения проблемы вовсе не означает невозможности ее решения для частных случаев. Прежде чем остановиться на этом вопросе, давайте посмотрим, что требуется для преобразования грамматики в LL(1)-форму.

1. *Устранение левой рекурсии*

Грамматика, содержащая левую рекурсию, не является LL(1)-грамматикой. Рассмотрим правила

$$A \to A\alpha \quad \text{(левая рекурсия в } A\text{)}$$
$$A \to a$$

Здесь a есть символ-предшественник для обоих вариантов нетермина-

ла A. Аналогично грамматика, содержащая левый рекурсивный цикл, не может быть LL(1)-грамматикой, например

$$A \to BC$$
$$B \to CD$$
$$C \to AE$$

Можно показать, что грамматику, содержащую левый рекурсивный цикл, нетрудно преобразовать в грамматику, содержащую только прямую левую рекурсию, и далее, за счет введения дополнительных нетерминалов, левую рекурсию можно исключить полностью (в действительности она заменяется правой рекурсией, которая не представляет проблемы в отношении LL(1)-свойства). Из теоремы о нормальной форме Грейбаха, упоминавшейся в разд. 4.1, вытекают два следствия.

В качестве примера обратимся к грамматике с порождающими правилами

$$S \to Aa \qquad C \to Dd$$
$$A \to Bb \qquad C \to e$$
$$B \to Cc \qquad D \to Az$$

которая имеет левый рекурсивный цикл, вовлекающий A, B, C, D. Чтобы заменить этот цикл на прямую левую рекурсию, мы можем действовать следующим образом. Упорядочим нетерминалы, а именно S, A, B, C, D. Рассмотрим все порождающие правила вида

$$X_i \to X_j\gamma,$$

где X_i и X_j — нетерминалы, а γ — строка терминальных и нетерминальных символов. В отношении правил, для которых $j \geqslant i$, никакие действия не производятся. Однако это неравенство не выдерживается для всех правил, если есть левый рекурсивный цикл. При выбранном нами порядке мы имеем дело с единственным правилом:

$$D \to Az$$

так как A *предшествует* D в этом упорядочении. Теперь начнем замещать A, пользуясь всеми правилами, имеющими A в левой части. В результате получаем

$$D \to Bbz$$

Поскольку B предшествует D в упорядочении, процесс повторяется, что дает правило:

$$D \to Ccbz$$

Затем он повторяется еще раз и дает два правила:

$$D \to ecbz$$
$$D \to Ddcbz$$

Теперь преобразованная грамматика выглядит следующим образом:

$$S \to Aa \qquad C \to e$$
$$A \to Bb \qquad D \to ecbz$$
$$B \to Cc \qquad D \to Ddcbz$$
$$C \to Dd$$

Все эти порождающие правила имеют требуемый вид, а левый рекурсивный цикл заменен прямой левой рекурсией.

Чтобы исключить прямую левую рекурсию, введем новый нетерминальный символ Z и заменим правила

$$D \to ecbz$$
$$D \to Ddcbz$$

на

$$D \to ecbz$$
$$D \to ecbzZ$$
$$Z \to dcbz$$
$$Z \to dcbzZ$$

Заметим, что до и после преобразования D генерирует регулярное выражение

$$(ecbz)\ (dcbz)^*$$

Обобщая, можно показать, что если нетерминал A появляется в левых частях $r+s$ порождающих правил, r из которых используют прямую левую рекурсию, а s — нет, т. е.

$$A \to A\alpha_1,\ A \to A\alpha_2, \cdots, A \to A\alpha_r$$
$$A \to \beta_1,\ A \to \beta_2, \cdots,\ A \to \beta_s$$

то эти правила можно заменить на следующие [27]:

$$\left.\begin{array}{l} A \to \beta_i \\ A \to \beta_i Z \end{array}\right\}\ 1 \leqslant i \leqslant s \qquad \left.\begin{array}{l} Z \to \alpha_i \\ Z \to \alpha_i Z \end{array}\right\}\ 1 \leqslant i \leqslant r$$

Неформальное доказательство заключается в том, что до и после преобразования A генерирует регулярное выражение

$$(\beta_1 | \beta_2 | \cdots | \beta_s)\ (\alpha_1 | \alpha_2 | \cdots | \alpha_r)^*.$$

Другой метод исключения левой рекурсии (по Фостеру) приводится в [22].

Левую рекурсию всегда можно исключить из грамматики, и нетрудно написать программу для общего случая.

2. *Факторизация*

Во многих ситуациях грамматики, не обладающие признаком LL(1), можно преобразовать в LL(1)-грамматики с помощью процесса факторизации. Пример такой ситуации мы видели в начале разд. 4.2. Аналогичным образом, порождающие правила

$$S \to aSb$$
$$S \to aSc$$
$$S \to \varepsilon$$

можно преобразовать путем факторизации в правила

$$S \to aSX$$
$$S \to \varepsilon$$
$$X \to b$$
$$X \to c$$

и полученной в результате грамматикой будет LL(1).

Процесс факторизации, однако, нельзя автоматизировать, распространив его на общий случай. Иначе это противоречило бы неразрешимости проблемы определения наличия признака LL(1) у языка. Следующий пример показывает, что может произойти.

Рассмотрим правила

1. $P \to Qx$
2. $P \to Ry$
3. $Q \to sQm$
4. $Q \to q$
5. $R \to sRn$
6. $R \to r$

Оба множества направляющих символов для двух вариантов P содержат s, и, пытаясь «вынести s за скобки», мы замещаем Q и R в правых частях правил 1 и 2:

$$P \to sQmx$$
$$P \to qx$$
$$P \to sRny$$
$$P \to ry$$

Эти правила можно заменить следующими:

$$P \to qx$$
$$P \to ry$$
$$P \to sP_1$$
$$P_1 \to Qmx$$
$$P_1 \to Rny$$

Правила для P_1 аналогичны первоначальным правилам для P и имеют пересекающиеся множества направляющих символов. Мы можем преобразовать эти правила так же, как и правила для P:

$$P_1 \to sQmmx$$
$$P_1 \to qmx$$
$$P_1 \to sRnny$$
$$P_1 \to rny$$

Факторизуя, получаем

$$P_1 \to qmx$$
$$P_1 \to rny$$
$$P_1 \to sP_2$$
$$P_2 \to Qmmx$$
$$P_2 \to Rnny$$

Правило для P_2 аналогично правилам для P_1 и P, но длиннее их, и теперь нам уже очевидно, что этот процесс бесконечный. Все попытки преобразовать грамматику в LL(1)-форму с помощью какого-либо алгоритма при некоторых входах обречены на неудачу (или зацикливание). Чем замысловатее (и сложнее) алгоритм, тем больше случаев он может охватить, однако всегда найдутся какие-либо входы, с которыми он потерпит поражение. Как же тогда разработчик компилятора найдет LL(1)-грамматику, на которой он будет базировать свой синтаксический анализатор для компилируемого языка? Вероятно, ему придется воспользоваться каким-нибудь (несовершенным) средством преобразования имеющейся у него грамматики. Примером такого средства служит SID (схема улучшения синтаксиса) Фостера [19]. Хотя SID и «несовершенен», он стал неоценимым для многих разработчиков компиляторов (включая автора) в Великобритании и других странах благодаря своей способности преобразовывать множество грамматик в LL(1)-форму.

Возникает вопрос, что же делать разработчику компиляторов, если его преобразователь грамматики не может выдать LL(1)-грамматику? В этом случае он должен попытаться определить на основании выхода преобразователя, какая часть грамматики не поддается преобразованию, и либо преобразовать эту часть вручную в LL(1)-форму, либо переписать ее так, чтобы преобразователь грамматики смог с

ней совладать. Это возможно при условии, что используется LL(1)-язык. Если данное условие не выполняется, то синтаксический анализатор LL(1) нельзя получить, не изменив реализуемый язык. Выход преобразователя может указать, почему язык не является LL(1)-языком.

С точки зрения пользователя, применение типичного преобразователя грамматики может привести к четырем ситуациям в зависимости от того, является ли язык LL(1)-языком или нет и достаточно ли «умен» преобразователь, чтобы выполнить «правильное» действие для заданного входа, т. е. преобразовать грамматику в LL(1)-форму, если это возможно, или выдать соответствующее диагностическое сообщение в противном случае. Эти ситуации схематически представлены в табл. 4.11.

Таблица 4.11

LL(1)-язык	Язык, не являющийся LL(1)
Успешно преобразованная грамматика	Выдано соответствующее значение
Преобразователь зациклился или остановился, указав, почему нельзя преобразовать грамматику	Преобразователь зациклился

Верхняя часть таблицы соответствует более удовлетворительным выходам преобразователя, а нижняя — менее удовлетворительным. Чем хитроумнее программа преобразователя, тем чаще выход окажется удовлетворительным, но из-за неразрешимости этой проблемы для любого преобразователя найдутся такие входы, которые поставят его в тупик.

Что касается такого языка, как Алгол 68, то его синтаксис не должен представлять особых трудностей для хорошего преобразователя. Так, у одного преобразователя вызвал затруднение следующий случай, показанный на примере части синтаксиса для фактического описателя структуры:

STRAD→struct orb LIST crb
LIST→EL
LIST→EL comma LIST
EL→ad FIELDS
FIELDS→tag
FIELDS→tag comma FIELDS

где *ad* может быть **real** или **int**, *tag* служит идентификатором, а *orb* и *crb* обозначают литеры «(» и «)» соответственно. Типичная генерируемая строка:

struct (**int** *a, b*, **real** *x, y*)

Проблема заключается в двойном употреблении запятой (для разделения двух *EL* и двух *tag*). Этот синтаксис можно переписать таким образом, чтобы генерировался тот же язык, а запятая появлялась в правилах только один раз:

STRAD→struct orb LIST crb
LIST→ad tag TAIL

TAIL→comma ITEM TAIL
→ε
ITEM→ad tag
→tag

Приведенная грамматика является LL(1)-грамматикой, так что никаких дальнейших преобразований не требуется. Другими источниками затруднений при использовании Алгола 68 могут быть множество случаев употребления открывающих круглых скобок и некоторые аспекты форматов (см. разд. 5.5). Если контекстно-свободная грамматика для Алгола 68 (или, по меньшей мере, ее контекстно-свободные варианты) берется из пересмотренного сообщения, то с помощью нескольких простых преобразований вручную, подобных вышеописанному, хороший преобразователь грамматики выдаст LL(1)-грамматику для языка, на котором можно базировать синтаксический анализатор. Следует отметить, что почти все проблемы по разбору Алгола 68 (какой бы метод ни применялся) возникают в результате перегрузки символов, в основном открывающей круглой скобкой и запятой.

4.5. LL(1)-ТАБЛИЦЫ РАЗБОРА

Найдя LL(1)-грамматику для языка, можно перейти к следующему этапу — применению найденной грамматики для приведения в действие в компиляторе фазы разбора. Этот этап аналогичен рекурсивному спуску, только здесь исключаются многочисленные вызовы процедур благодаря представлению грамматики в табличном виде (таблицы разбора) и использованию не зависящего от исходного языка модуля компилятора для проведения по таблице во время чтения исходного текста. Модуль компилятора, который мы будем называть драйвером, всегда указывает на то место в синтаксисе,которое соответствует текущему входному символу. Драйверу требуется стек для запоминания адресов возврата всякий раз, когда он входит в новое порождающее правило, соответствующее какому-либо нетерминалу. Представление синтаксиса должно быть таким, чтобы обеспечить эффективность синтаксического анализатора в отношении скорости работы.

Опишем сначала возможный вид таблицы разбора, довольно простой и понятный, а затем рассмотрим способы ее оптимизации относительно объема и т. д.

Таблица разбора представляется как одномерный массив структур:

```
mode parsel=struct (list terminals,
     int jump, bool accept, stack, return, error)
```

где

```
mode list=struct (string term, ref list next)
```

Сама таблица описывается как

```
[1 : ptsize]parsel pt
```

Нам также нужен стек для адресов возврата и указатель стека:

```
flex[0 : 25]int stack
int Sptr := 0;
```

В таблице каждому шагу процесса разбора соответствует один элемент. В процессе разбора осуществляется целый ряд различных шагов, а именно:

1. Проверка предварительно просматриваемого символа с тем, чтобы выяснить, не является ли он направляющим для какой-либо конкретной правой части порождающего правила. Если этот символ — не направляющий и имеется альтернативная правая часть правила, то она проверяется на следующем этапе. В особом случае, когда правая часть начинается с терминала, множество направляющих символов состоит только из одного этого терминала.

2. Проверка терминала, появляющегося в правой части порождающего правила.

3. Проверка нетерминала. Она заключается в проверке нахождения предварительно просматриваемого символа в одном из множеств направляющих символов для данного нетерминала, помещении в стек адреса возврата и переходу к первому правилу, относящемуся к этому нетерминалу. Если нетерминал появляется в конце правой части правила, то нет необходимости помещать в стек адрес возврата.

Таким образом, в таблицу разбора включается по одному элементу на каждое правило грамматики и на каждый экземпляр терминала или нетерминала правой части правила. Кроме того, в таблице будут находиться элементы на каждую реализацию пустой строки в правой части правила (по одному на каждую реализацию).

Драйвер содержит цикл процедуры, тело которой обрабатывает элемент таблицы разбора и определяет следующий элемент для обработки. Поле перехода обычно дает следующий элемент обработки, если значение поля возврата не окажется истиной. В последнем случае адрес следующего элемента берется из стека (что соответствует концу правила). Если же предварительно просматриваемый символ отсутствует в списке терминалов и значение поля ошибки окажется ложью, нужно обрабатывать следующий элемент таблицы с тем же предварительно просматриваемым символом (способ обращения с альтернативными правыми частями правил).

В приводимой ниже процедуре *la* представляет собой логическое значение, которое определяет, надо ли считывать новый предварительно просматриваемый символ до обработки следующего элемента таблицы разбора. Например, *la* имеет значение **false**, когда предварительно просматриваемый символ не является направляющим для какой-либо конкретной правой стороны правила, и требуется исследовать множество направляющих символов, соответствующее другой правой части. Если предварительно просматриваемый символ не содержится в текущем множестве направляющих символов и поле ошибки **parsel** будет **true**, то выдается сообщение о синтаксической ошибке. Если поле стека обрабатываемой **parsel** имеет значение **true**, то до перехода к адресу, задаваемому полем перехода, в стек помещается адрес следующей **parsel**.

Вызов процедуры разбора *parse (pt)* приведет в действие драйвер, где

```
proc parse = ([ ] parsel pt) bool:
begin int i := 1;
  push 0;
  bool la := true;
  string lookahead;
  while if la then lookahead := lexread fi;
  lookahead ≠ "⊥" and i ≠ 0
  do if lookahead oneof terminals of pt [i]
```

```
        then la := accept of pt [i];
          if return of pt [i] then i := pop
          else if stack of pt [i] then push (i + 1) fi;
            i := jump of pt [i]
          fi
        elif error of pt [i] then
          syntax error
        else i plusab 1; la := false
        fi
      od;
      i = 0 and lookahead = " ⊥"
    end
```

«⊥» — специальный символ, появляющийся после окончания предложения. Процедуры *push* и *pop* и оператор ***oneof*** определяются следующим образом:

```
proc push = (int x) void:
begin if Sptr > upb stack then stack plusab 10 fi;
  stack [Sptr] := x;
  Sptr plusab 1
end;
proc pop = int:
begin Sptr minusab 1;
  if Sptr < 0 then stack empty fi;
  stack [Sptr]
end;
prio oneof = 1;
op oneof = (string la, ref list l) bool:
begin bool result := false; list ll := l;
  while (ll inst nil) and not result
  do if la = term of ll
    then result := true
    else ll := next of ll
    fi
  od;
  result
end
```

nil обозначает конец списка на Алголе 68, а **plusab** определяется (помимо его стандартного значения) как:

```
op plusab = (ref flex [ ] int S, int i) ref flex [ ] int:
begin [0: upb S + i] int temp;
 temp [0: upb S] := S;
 for j from upb S + 1 to upb temp
 do temp [j] := 0
 od;
 S := temp
end
```

Кроме того, существуют процедуры обращения с подобными ситуациями — *syntax error* и *stack empty*, а *lexread* выдает строку, соответствующую одному символу языка.

Драйвер совершенно не зависит от разбираемого языка и может использоваться в целом ряде компиляторов. Он относительно мал, так что бóльшая часть объема памяти, занимаемого синтаксическим ана-

лизатором, приходится на таблицу разбора, размер которой пропорционален размеру грамматики (языка). Поэтому для «больших» языков размер анализатора, грубо говоря, пропорционален размеру грамматики. В качестве примера выведем таблицу разбора для следующей грамматики:

(1) *PROGRAM→begin DECLIST comma STATELIST end*
(2) *DECLIST→d X*
(3) *X→semi DECLIST*
(4) *X→ε*
(5) *STATELIST→s Y*
(6) *Y→semi STATELIST*
(7) *Y→ε*

Сначала представим грамматику в виде схемы, показанной на рис. 4.2. В скобках слева и справа на рисунке указаны номера соответствующих элементов таблицы разбора. На основании этой схемы можно построить таблицу разбора (табл. 4.12). *0* появляется в поле перехода, когда оно не имеет отношения к делу.

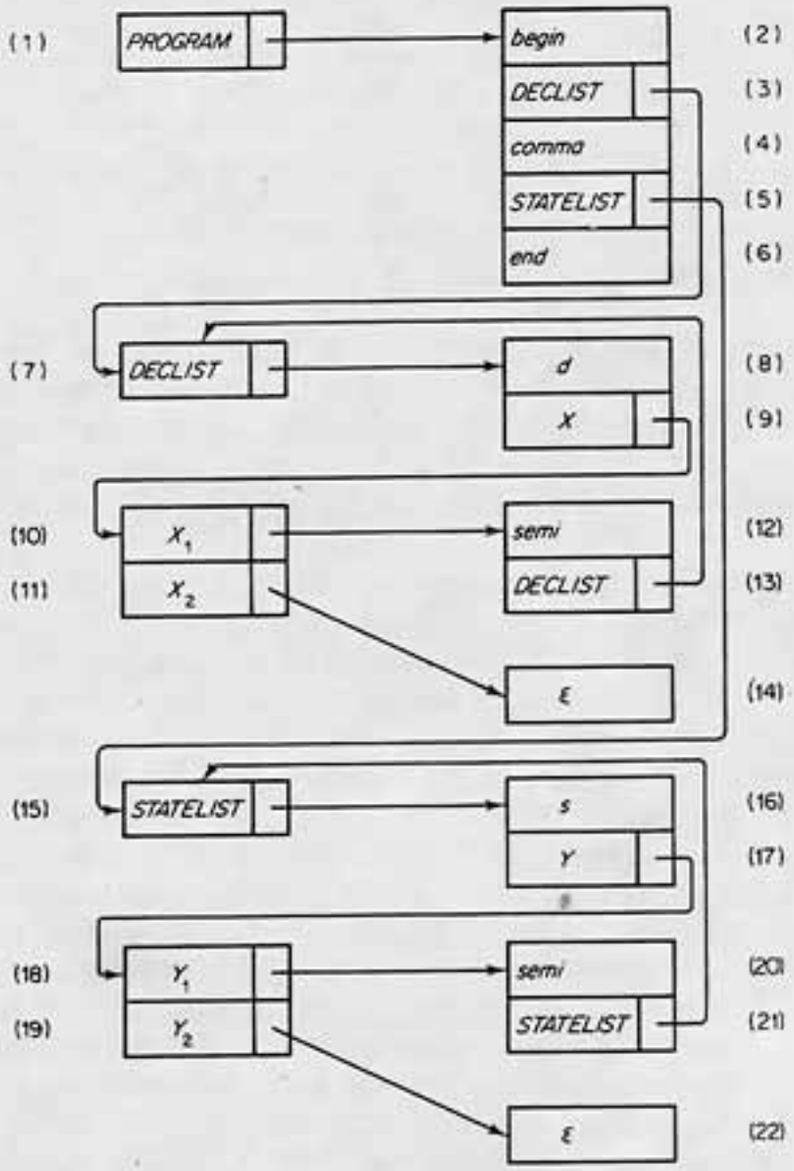


Рис. 4.2.

Таблица 4.12

	terminals	*jump*	*accept*	*stack*	*return*	*error*
1	{*begin*}	*2*	**false**	**false**	**false**	**true**
2	{*begin*}	*3*	**true**	**false**	**false**	**true**
3	{*d*}	*7*	**false**	**true**	**false**	**true**
4	{*comma*}	*5*	**true**	**false**	**false**	**true**
5	{*s*}	*15*	**false**	**true**	**false**	**true**
6	{*end*}	*0*	**true**	**false**	**true**	**true**
7	{*d*}	*8*	**false**	**false**	**false**	**true**
8	{*d*}	*9*	**true**	**false**	**false**	**true**
9	{*semi, comma*}	*10*	**false**	**false**	**false**	**true**
10	{*semi*}	*12*	**false**	**false**	**false**	**false**
11	{*comma*}	*14*	**false**	**false**	**false**	**true**
12	{*semi*}	*13*	**true**	**false**	**false**	**true**
13	{*d*}	*7*	**false**	**false**	**false**	**true**
14	{*comma*}	*0*	**false**	**false**	**true**	**true**
15	{*s*}	*16*	**false**	**false**	**false**	**true**
16	{*s*}	*17*	**true**	**false**	**false**	**true**
17	{*semi, end*}	*18*	**false**	**false**	**false**	**true**
18	{*semi*}	*20*	**false**	**false**	**false**	**false**
19	{*end*}	*22*	**false**	**false**	**false**	**true**
20	{*semi*}	*21*	**true**	**false**	**false**	**true**
21	{*s*}	*15*	**false**	**false**	**false**	**true**
22	{*end*}	*0*	**false**	**false**	**true**	**true**

Рассмотрим предложение (со следующим символом «⊥»)

begin d semi d comma s semi s end ⊥

Ниже приводится разбор этого предложения.

Заметим, что некоторые терминалы проверяются несколько раз. Такой неоднократной проверки, очевидно, можно избежать, если разработчики согласны отложить обнаружение некоторых синтаксических ошибок на более поздний этап («поздний» в смысле шагов разбора, но не считанного текста). Можно также сократить число элементов в таблице разбора.

Желательно, чтобы элементы в таблице разбора были небольшими. Тогда, ограничив число терминалов в грамматике и диапазон переходов в пределах таблицы разбора, можно будет, вероятно, упаковать каждый элемент таблицы в машинное слово (24 бита в машинах серии ICL 1900). Это, очевидно, замедлит работу синтаксического анализатора во время компиляции из-за распаковки, которую придется выполнять, но зато позволит получить небольшую таблицу разбора (на несколько тысяч слов) даже для таких объемных языков, как Алгол 68. Как мы увидим позднее, при обсуждении методов разбора снизу вверх размер таблицы разбора часто представляет бо́льшую проблему для разработчика компиляторов, чем скорость анализатора.

При наличии LL(1)-грамматики можно написать программу для получения соответствующей таблицы разбора. Используемый при этом алгоритм имеет много общего с проверкой наличия признака LL(1), и их часто объединяют, как, например, в пакете SID/SAG, разработанном в RSRE в Малверне. Драйвер можно, конечно, применять многократно для различных компиляторов, так что при наличии соответствующих средств программного обеспечения возможно получение LL(1)-анализатора из грамматики с наименьшей затратой усилий.

i	Действие	Стек разбора
1.	*begin* считывается и проверяется; перейти к *pt* [2]	
2.	*begin* считывается и принимается; перейти к *pt* [3]	
3.	*d* считывается и проверяется; 4 помещается в стек; перейти к *pt* [7]	
7.	*d* проверяется; перейти к *pt* [8]	
8.	*d* проверяется и принимается; перейти к *pt* [9]	
9.	*semi* считывается и проверяется; перейти к *pt* [10]	
10.	*semi* проверяется; перейти к *pt* [12]	
12.	*semi* проверяется и принимается; перейти к *pt* [13]	
13.	*d* считывается и проверяется; перейти к *pt* [7]	
7.	*d* проверяется; перейти к *pt* [8]	
8.	*d* проверяется и принимается; перейти к *pt* [9]	
9.	*comma* считывается и проверяется; перейти к *pt* [10]	

Продолжение

i	Действие	Стек разбора
10.	*comma* не совпадает с *semi*; ошибка ложь — перейти к *pt* [11]	4 0
11.	*comma* проверяется; перейти к *pt* [14]	4 0
14.	*comma* проверяется; возврат истина — *pop* 4; перейти к *pt* [4]	0
4.	*comma* проверяется и принимается; перейти к *pt* [5]	0
5.	*s* считывается и проверяется; 6 помещается в стек; перейти к *pt* [15]	6 0
15.	*S* проверяется; перейти к *pt* [16]	6 0
16.	*s* проверяется и принимается; перейти к *pt* [17]	6 0
17.	*semi* считывается и проверяется; перейти к *pt* [18]	6 0
18.	*semi* проверяется; перейти к *pt* [20]	6 0
20.	*semi* проверяется и принимается; перейти к *pt* [21]	6 0
21.	*s* считывается и проверяется; перейти к *pt* [15]	6 0
15.	*s* проверяется; перейти к [*pt*] 16	6 0

Продолжение

i	Действие	Стек разбора
16	*s* проверяется и принимается; перейти к *pt* [17]	*6* *0*
17.	*end* считывается и проверяется; перейти к *pt* [18]	*6* *0*
18.	*end* не совпадает с *semi*; ошибка ложь — перейти к *pt* [19]	*6* *0*
19.	*end* проверяется; перейти к *pt* [22]	*6* *0*
22.	*end* проверяется; возврат истина — *pop* 6; перейти к *pt* [6]	*0*
6.	*end* проверяется и принимается; возврат истина — *pop 0; i* : *=0*	
0	Разбор заканчивается	

LL(1)-метод разбора имеет ряд преимуществ:

1. Никогда не требуется возврат, поскольку этот метод — детерминированный.

2. Время разбора (приблизительно) пропорционально длине программы. Как будет показано ниже, для процесса компиляции в целом это может оказаться не так, поскольку некоторые действия, выполняемые во время компиляции, например поиск в таблице, могут потребовать различное время в зависимости от размера таблиц, что в свою очередь связано с размером программы.

3. Имеются хорошие диагностические характеристики и существует возможность исправления ошибок, так как синтаксические ошибки распознаются по первому неприемлемому символу, а в таблице разбора есть список возможных символов продолжения. Методы исправления ошибок обсуждаются в гл. 12.

4. Таблицы разбора меньше, чем соответствующие таблицы в других методах разбора.

5. LL(1)-разбор применим к широкому классу языков — всех языков, имеющих LL(1)-грамматики. Нужно, однако, добавить, что в боль-

шинстве случаев «очевидной» грамматикой для языка программирования оказывается не LL(1), и ее придется преобразовать в LL(1)-грамматику до того, как будет применен этот метод. LR(1)-разбор, как мы увидим в следующей главе, можно применять к еще более широкому классу языков, а преобразования грамматик требуются редко. В той же главе мы рассмотрим и относительные достоинства этих двух методов.

Мы можем расширить принцип LL(1)-грамматик до LL(k)-грамматик и, чтобы различать альтернативные порождающие правила для нетерминалов, использовать $k(k>1)$ предварительно просматриваемых символов. Тогда анализатору придется рассматривать строки длиной k, а так как это весьма вероятно, задача станет намного сложнее. На практике для разбора даже LL(2)-грамматики используются редко, хотя грамматика, частично являющаяся LL(2)-грамматикой, могла бы применяться на основе LL(1)-анализатора, которому разрешается при необходимости исследовать дополнительный символ.

Примером LL(2)-грамматики может служить грамматика, приведенная в начале разд. 4.2, а примером LL(3)-грамматики — представленная ниже:

PROGRAM→*begin DECLIST semi STATELIST end*
DECLIST→*d semi DECLIST*
d
STATELIST→*s semi STATELIST*
s

Эквивалентная LL(1)-грамматика:

PROGRAM→*begin d semi X end*
X→*d semi X*
X→*s Y*
Y→*e*
Y→*semi s Y*

Упражнения

4.1. Определите аргументированно, какие из следующих языков являются детерминированными:

а) $\{\alpha c \alpha^r | \alpha$ в $\{0,1\}^*\}$;
б) $\{c\alpha\alpha^r | \alpha$ в $\{a,b\}^*\}$
(α^r означает обратную α);
в) $\{0^n 1^n | n>0\}$;
г) $\{0^n 1^{2n} | n>0\}$;
д) $\{0^n 1^n | n>0\} \cup \{0^n 1^{2n} | n>0\}$

4.2. Покажите, что следующие языки не являются контекстно-свободными:

а) $\{a^n b^n c^n | n>0\}$;
б) $\{a^i | i$ есть простое число$\}$.

4.3. Напишите синтаксический анализатор рекурсивного спуска, соответствующий грамматике со следующими порождающими правилами (E — начальный символ):

$E \to TG$
$G \to +TG$
e
$T \to FU$
$U \to \times FU$
e
$F \to (E)$
identifier

4.4. Контекстно-свободная грамматика называется q-грамматикой тогда (и только тогда), когда она обладает следующими свойствами:
1. Правая часть каждого правила либо начинается с терминала, либо пустая.
2. Для каждого нетерминала, появляющегося в левой части более чем одного правила, множества направляющих символов, соответствующих правым частям альтернативных правил, являются непересекающимися.

Докажите, что
а) любая q-грамматика есть LL(1);
б) любая s-грамматика есть q-грамматика;
в) любую LL(1)-грамматику можно преобразовать в q-грамматику.

4.5. Приведите LL(1)-грамматики для каждого из следующих языков:
1. $\{0^n a 1^{2n} \mid n \geq 0\}$.
2. $\{\alpha \mid \alpha$ находится в $\{0,1\}^*$ и не содержит две последовательные $1\}$.
3. $\{\alpha \mid \alpha$ состоит из равного числа нулей и единиц$\}$.

4.6. Преобразуйте грамматику со следующими правилами в LL(1)-форму (E — начальный символ):

$$\begin{array}{ll} E \to E+T & F \to (E) \\ E \to T & F \to x \\ T \to T \times F & F \to y \\ T \to F & \end{array}$$

4.7. Обладает ли грамматика со следующими правилами признаком LL(1) или нет? Обоснуйте свой ответ.

$$\begin{array}{ll} S \to AB & C \to \varepsilon \\ S \to PQx & P \to pP \\ A \to xy & P \to \varepsilon \\ A \to m & Q \to qQ \\ B \to bC & Q \to \varepsilon \\ C \to bC & \end{array}$$

(S — начальный символ).

4.8. Докажите, что все LL(1)-грамматики являются однозначными.

4.9. Постройте LL(1)-таблицу разбора для языка, определяемого посредством LL(1)-грамматики с правилами:

$$\begin{array}{l} PROGRAM \to begin\ d\ semi\ X\ end \\ X \to d\ semi\ X \\ X \to s\ Y \\ Y \to \varepsilon \\ Y \to semi\ s\ Y \end{array}$$

4.10. Постройте LL(1)-таблицу разбора для языка, определяемого грамматикой из упр. 4.6.

Глава 5

СИНТАКСИЧЕСКИЙ АНАЛИЗ СНИЗУ ВВЕРХ

В этой главе мы рассмотрим методы разбора снизу вверх в целом и опишем один конкретный метод (LR-разбор), который применим к широкому классу языков и грамматик. Мы покажем, как автоматически построить синтаксический анализатор на основании LR-грамматики, и дадим сравнительную оценку достоинств LL- и LR-методов разбора (в последнем разделе).

5.1. РАЗБОР СНИЗУ ВВЕРХ

Как уже отмечалось, синтаксические анализаторы, работающие по принципу «снизу вверх», сводят предложение языка к начальному символу путем последовательного применения правил грамматики. Рассмотрим, например, язык, генерируемый правилами

(1) *S*→**real** *IDLIST*
(2) *IDLIST*→*IDLIST*, *ID*
(3) *IDLIST*→*ID*
(4) *ID*→A|B|C|D

(для простоты примем (4) за одно порождающее правило).

Предложение

real *A*, *B*, *C*

принадлежит этому языку и может быть разобрано снизу вверх следующим образом. Каждый символ, считанный анализатором, немедленно помещается в верхнюю часть стека анализатора. Например, считывается **real** и помещается в стек. Первые два действия анализатора схематически показываются так:

real ↑ *A*, *B*, *C*

real *A* ↑ , *B*, *C*

Здесь стрелка ставится непосредственно после последнего считанного символа. Затем анализатор заменяет *A* с помощью правила (4). Это действие известно как приведение. В разборе снизу вверх правила применяются иначе, чем в разборе сверху вниз, т. е. правые части правил заменяются их соответствующими левыми частями. Представим приведение схематически:

real $A_{\uparrow}$, *B*, *C*

Далее применяется правило (3) для выполнения другого приведения:

real $A_{\uparrow}$, *B*, *C*

Теперь анализатор читает следующий символ:

real *A*, $_{\uparrow}$ *B*, *C*

и другой:

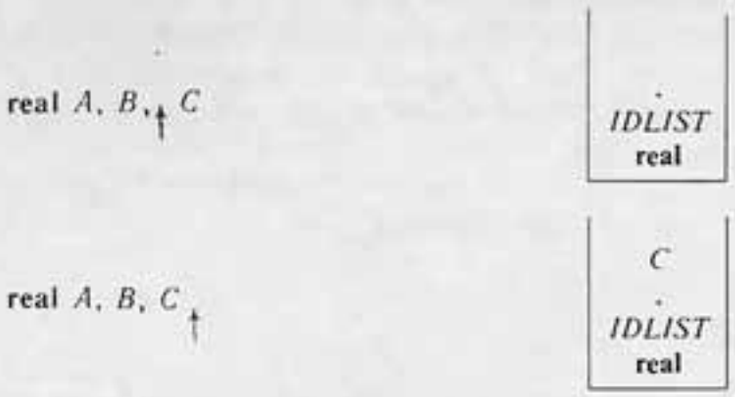
real *A*, *B*$_{\uparrow}$, *C*

Очередные два действия — приведения с использованием правил (4) и (2):

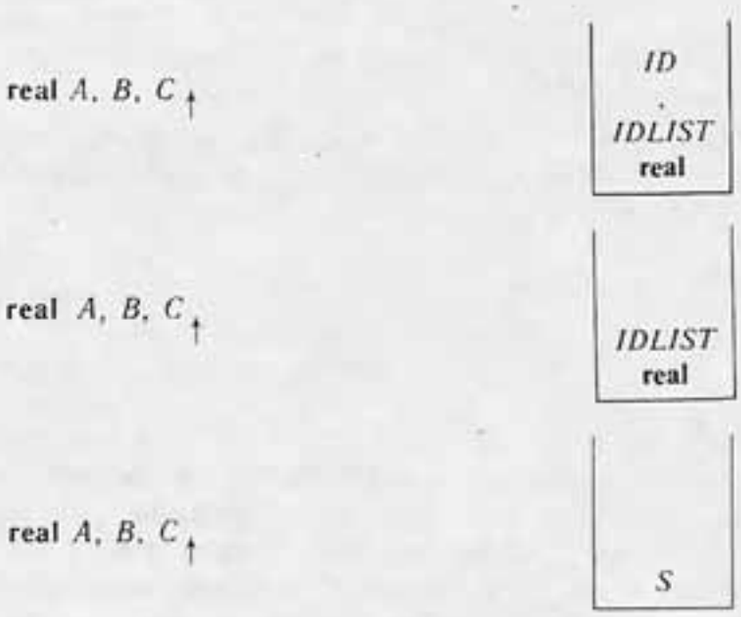
real *A*, *B*$_{\uparrow}$, *C*

real *A*, *B*$_{\uparrow}$, *C*

и последующим считыванием еще двух символов:



Последние три действия — приведения с использованием правил (4), (2) и (1):



Разбор считается завершенным, когда в стеке остается только начальный символ и предложение считано целиком. Стек разбора соответствует части автомата магазинного типа, описанного в гл. 4.

Синтаксический анализатор, работающий по принципу «снизу вверх», выполняет действия двух типов:

1) *сдвиг*, во время которого считывается и помещается в стек символ. Это соответствует продвижению на один пункт вдоль какого-либо правила грамматики;

2) *приведение*, во время которого множество элементов в верхней части стека замещаются каким-либо нетерминалом грамматики с помощью одного из порождающих правил этой грамматики.

Методы разбора снизу вверх почти всегда детерминированные, и перед анализатором стоит единственная задача: знать в конкретной

ситуации, какое действие (сдвиг или приведение) выполнять. Если возможны два разных приведения, анализатор должен знать, какое из них выполнять и выполнять ли их вообще. Конечно, необходимое условие для осуществления приведения заключается в том, чтобы правая часть какого-либо правила появилась в вершине стека. Но это условие, однако, не всегда является достаточным, как видно из вышеприведенного примера, где было бы неправильным выполнять приведение, пользуясь правилом (1) в ситуации

real $A_{\uparrow}$, B, C

Хотя это не относится к ситуации

real A, B, $C_{\uparrow}$

Допускается также, чтобы правые части более чем одного правила появились в вершине стека, например, в ситуации

real A, $B_{\uparrow}$, C

Здесь правильным было бы выполнять приведение с помощью правила (2) и неправильным — с помощью правила (3).

Детерминированный метод разбора снизу вверх предлагает некоторый критерий для выбора решения в случае возникновения подобных конфликтов. Так, решение можно принять на основании одного или более предварительно просмотренных символов (как в LL-разборе) или на основании информации, имеющейся в стеке разбора. Если считать, что за предложением в вышеприведенном примере следует, например, знак окончания «⊥», то когда стек содержит

решение о сдвиге или приведении посредством правила (1) определяется тем, каким будет предварительно просматриваемый символ: «,» или «⊥».

Некоторые методы разбора снизу вверх, такие, как методы с предшествованием и с ограниченным контекстом [20], могут применяться только к относительно небольшому подмножеству языков. Наиболее универсальным является LR-метод, поскольку он применим ко всем

языкам (и грамматикам), разбираемым детерминированно. LR(k)-грамматикой называется грамматика, при использовании которой в качестве основы для анализатора снизу вверх все конфликты типа сдвиг/приведение и приведение/приведение можно разрешать на основании уже прочитанного текста и *фиксированного* числа предварительно просматриваемых символов (максимум k). Буква L в LR показывает, что строки читаются слева направо, а R — что получается правосторонний разбор. Теперь нам ясно, что правила, используемые в правостороннем разборе, который обычно ассоциируется с *методом разбора снизу вверх*, перечисляются в порядке, *обратном* порядку при продвижении от начального символа предложения вправо (см. разд. 2.4).

При первоначальном описании LR-разбора [32] этот метод казался довольно громоздким и неэффективным. Однако в последующих работах [1], [16] и др. он стал более привлекательным с практической точки зрения.

5.2. LR(1)-ГРАММАТИКИ И ЯЗЫКИ

В предыдущем разделе мы объяснили, что подразумевается под LR(k)-грамматикой. На практике k всегда принимает значение 0 или 1. Рассмотрение множеств даже двух предварительно просматриваемых символов при разрешении конфликтов сделало бы метод крайне громоздким. Более того, мы не смогли бы разбирать никакие другие языки с помощью этого метода, так как можно показать, что любой LR(k)-язык является также LR(1)-языком (т. е. может генерироваться LR(1)-грамматикой) и даже LR(0)-языком, если допустить, что за каждым предложением следует знак окончания ([27], с. 185). Поэтому, хотя и существуют LR(2)-грамматики (но не LR(1))-грамматики, например

(1) $S \rightarrow F, S$
(2) $S \rightarrow F$
(3) $F \rightarrow a\ L$
(4) $L \rightarrow t$
(5) $L \rightarrow t, L$

LR(2)-языков (но не LR(1)-языков) не существует. Этот вариант совершенно не похож на вариант с LL, где, увеличивая значение k, всегда можно (хотя и не всегда практически оправданно) разбирать большее число языков. Грамматика с перечисленными выше порождающими правилами не является LR(1)-грамматикой, так как после считывания «ведущей» подстроки предложения at, при наличии предварительно просматриваемого символа «,», анализатор не может определить, что выполнять приведение с помощью правила (4) — или сдвиг. Для разрешения конфликта можно воспользоваться двумя предварительно просматриваемыми символами. Например, пара символов «,t» покажут сдвиг, а пара «,a» — приведение.

Однако по теории должна существовать LR(1)-грамматика, генерирующая тот же язык. Возможно, ею окажется такая грамматика:

$$S \rightarrow a\ t \mid F$$
$$F \rightarrow , ITEM\ F \mid \varepsilon$$
$$ITEM \rightarrow a \mid t$$

Приняв, что «,» в грамматике должна появляться только один раз, можно снять признак непринадлежности этой грамматики к клас-

су LR(1). Однако вышеприведенная грамматика показывает, что одних только предварительно просматриваемых символов не всегда достаточно, чтобы разрешить конфликты. В некоторых случаях нужно также принять во внимание левый контекст. Например, решение о том, нужно ли последовательность

al с предварительно просматриваемым символом «,»

приводить к *ITEM* или нет, зависит от того, первая это реализация данной последовательности или нет. К счастью, как мы увидим далее, решение о выполнении приведения в LR-разборе, в общем, не требует полного просмотра левого контекста или содержимого стека.

Выше уже упоминалось о том, что LR-метод применим ко всем языкам, которые можно разбирать детерминированно. В частности, все LL(1)-грамматики и языки обладают признаком LR(1). Это объясняется тем, что решение о применении конкретного правила во время LL(1)-разбора базируется на замене нетерминала в сентенциальной форме и одном предварительно просматриваемом символе, в то время как для решения о выполнении приведения в случае LR(1) требуется поместить в стек всю правую часть правила и считать следующий предварительно просматриваемый символ. Таким образом, в случае LR(1) имеется по меньшей мере столько же информации, на которой можно основывать решение о применении конкретного правила, сколько ее имеется в случае LL(1). Это можно показать схематически:

LL(1) точка принятия решения LR(1) точка принятия решения

γ $\underset{\downarrow}{a}\alpha$ $\underset{\downarrow}{l}\beta$

$\gamma a \alpha l \beta$ представляет сентенциальную форму, непосредственно после которой в случае LL(1) и непосредственно перед которой в случае LR(1) применяется (допустим) правило $A \rightarrow a\alpha$.

Существуют также грамматики и языки, которые обладают признаком LR(1), но не обладают признаком LL(1). Например, в разд. 5.1 была приведена LR(1)-грамматика, которая не являлась LL(1)-грамматикой. Того, что она содержит левую рекурсию, оказалось достаточно, чтобы она не была LL(1). В LR-разборе левая рекурсия не создает проблем.

Поскольку все языки, которые можно разобрать детерминированно, имеют LR(0)-грамматику, можно было бы ожидать, что остальную часть главы мы посвятим LR(0)-грамматикам. Однако многие типичные свойства грамматик языков программирования относятся не к LR(0)-свойствам, а скорее к LR(1)-свойствам, так что, рассматривая LR(1)-грамматики, мы, как правило, можем избежать затруднений, связанных с преобразованиями грамматик, которые требуются почти всегда при использовании LL-метода.

Это — положительное качество LR-разбора, особенно если вспомнить о неразрешимости проблемы преобразования грамматик. В отношении же LR-грамматик справедливо следующее:

1. Можно решить, обладает ли грамматика признаком LR(k) для заданного k. В частности, можно решить, является ли грамматика LR(0)- или LR(1)-грамматикой.

2. Нельзя решить, существует ли такое значение k, для которого заданная грамматика будет LR(k)-грамматикой. Этот результат не имеет особого практического значения с точки зрения разбора, так как обычно LR(k)-грамматики не представляют для нас интереса, если $k>1$.

Как основа для синтаксического анализатора LR(1)-грамматики обладают некоторыми привлекательными свойствами. Их более универсальный характер и тот факт, что они редко требуют преобразований, дают им преимущество перед LL-грамматиками. В разд. 5.5 мы дадим сравнительную оценку этих двух типов грамматик с практической точки зрения. Пока же в следующих двух разделах рассмотрим создание LR-анализаторов.

5.3. LR-ТАБЛИЦЫ РАЗБОРА

В разд. 5.1 мы познакомились в целом с работой анализатора по принципу «снизу вверх». Не обсуждался лишь вопрос о том, как решить, выполнять ли конкретное приведение, когда правая часть правила появляется в вершине стека. В этой головоломке не достает таблицы разбора, которая позволяет в каждом случае принимать правильное решение. В этом разделе мы опишем форму таблицы (весьма отличающуюся от LL-таблицы) и объясним ее использование, а в следующем — покажем, как эта таблица образуется из грамматики. Правила грамматики таковы:

(1) *S*→**real** *IDLIST*
(2) *IDLIST*→*IDLIST*, *ID*
(3) *IDLIST*→*ID*
(4) *ID*→*A*|*B*|*C*|*D*

Таблица разбора представляет собой прямоугольную матрицу. Она состоит из столбцов для каждого терминала и нетерминала грамматики плюс знак окончания и строк, соответствующих каждому состоянию, в котором может находиться анализатор. Позднее мы поясним значение всех состояний. Здесь же отметим, что каждое состояние соответствует той позиции в порождающем правиле, которой достиг анализатор. Таблица разбора для грамматики из разд. 5.1 показана в табл. 5.1. Не зависящая от языка часть анализатора, или драйвер, как мы называли ее в предыдущей главе, использует два стека (см. упр. 5.10) — стек символов и стек состояний. Таблица разбора включает элементы четырех типов:

1. Элементы сдвига. Эти элементы имеют вид *S7* и означают: поместить в стек символов символ, соответствующий столбцу; поместить в стек состояний 7 и перейти к состоянию 7; если входной символ — терминал, принять его.

2. Элементы приведения. Они имеют вид *R*4 и означают: выполнить приведение с помощью правила (4), т. е. допустив, что *n* есть

Таблица 5.1

Состояние	*S*	*IDLIST*	*ID*	**real**	«,»	*A* *B* *C* *D*	«⊥»
1	*HALT*			*S2*			
2		*S5*	*S4*			*S3*	
3					*R4*		*R4*
4					*R3*		*R3*
5					*S6*		*R1*
6			*S7*			*S3*	
7					*R2*		*R2*

число символов в правой части правила (4); удалить n элементов из стека символов и n элементов из стека состояний, и перейти к состоянию, указанному в верхней части стека состояний. Нетерминал в левой части правила (4) нужно считать следующим входным символом.

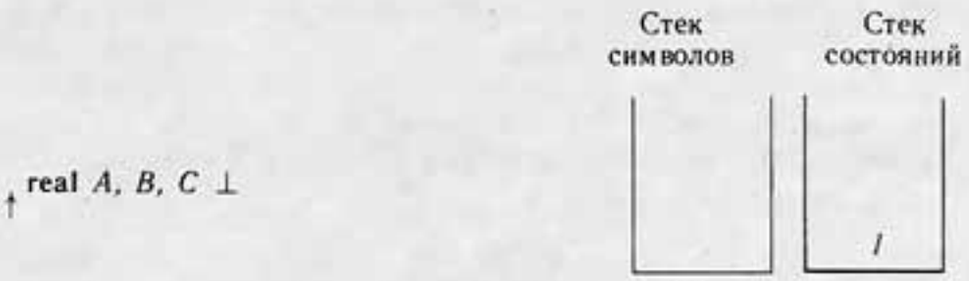
3. Элементы ошибок. Эти элементы являются пробелами в таблице и соответствуют синтаксическим ошибкам.

4. Элемент(ы) остановки. Ими завершается разбор.

Рассмотрим теперь, как с помощью вышеописанной таблицы разбора будет разбираться строка:

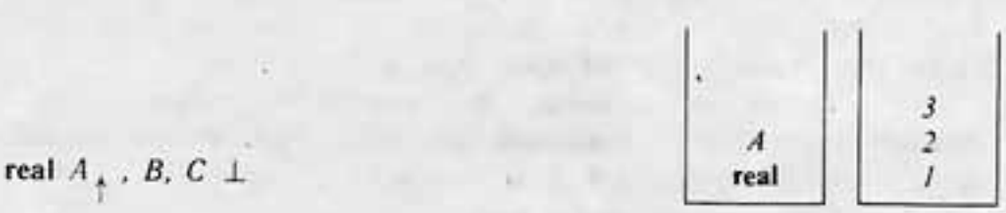
real *A, B, C* ⊥

Покажем содержимое стека символов и стека состояний на каждом этапе. Начнем с состояния *1*, которое отображается в стеке:



входной символ — **real** — из элемента таблицы (*1*, **real**), сдвиг в состояние *2*:

входной символ — *A*, сдвиг в состояние *3*:

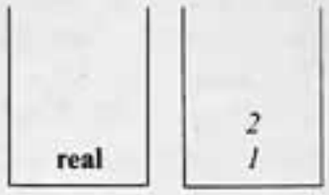


входной символ — «,», приведение посредством правила (4):

входной символ — *ID*, сдвиг в состояние *4*:

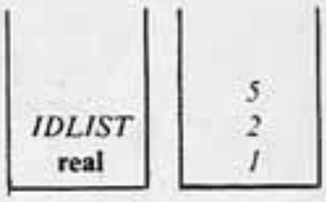
входной символ — «,», приведение посредством правила (3):

real $A_{\uparrow}$, B, C $\perp$



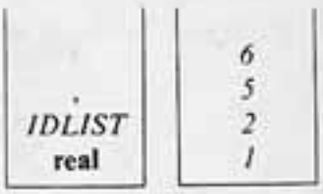
входной символ — *IDLIST*, сдвиг в состояние *5*:

real $A_{\uparrow}$, B, C



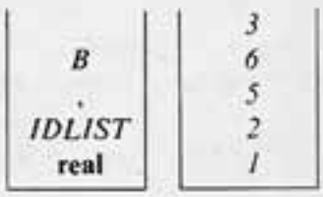
входной символ — «,», сдвиг в состояние *6*:

real A, $_{\uparrow}$ B, C $\perp$



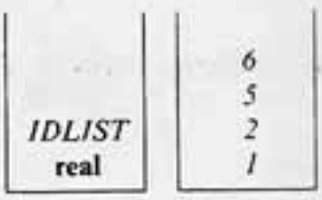
входной символ — *B*, сдвиг в состояние *3*:

real A, $B_{\uparrow}$, C $\perp$



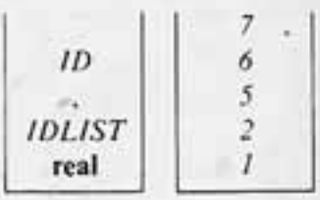
входной символ — «,», приведение посредством правила (4):

real A, $B_{\uparrow}$, C $\perp$



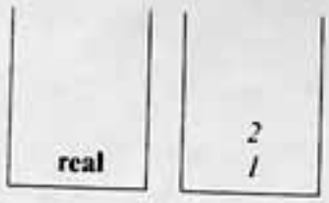
входной символ — *ID*, сдвиг в состояние *7*:

real A, $B_{\uparrow}$, C $\perp$



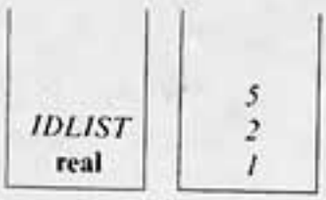
входной символ — «,», приведение посредством правила (2):

real *A*, *B*, *C* ⊥



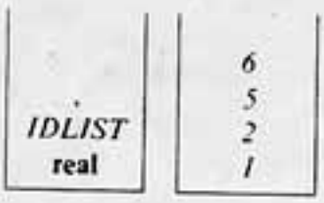
входной символ — *IDLIST*, сдвиг в состояние *5*:

real *A*, *B*, *C* ⊥



входной символ — «,», сдвиг в состояние *6*:

real *A*, *B*, *C* ⊥

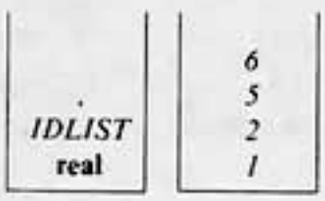


входной символ — *C*, сдвиг в состояние *3*:

real *A*, *B*, *C* ⊥

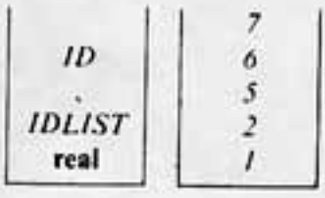
входной символ — «⊥», приведение посредством правила (4):

real *A*, *B*, *C* ⊥



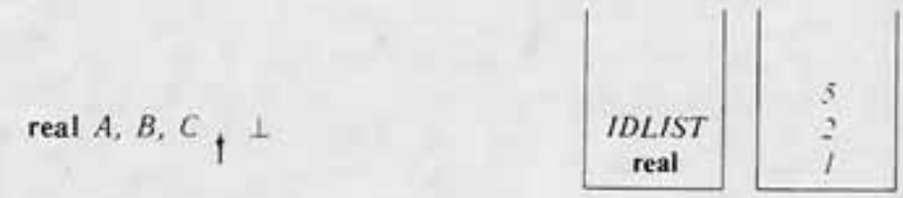
входной символ — *ID*, сдвиг в состояние *7*:

real *A*, *B*, *C* ⊥



входной символ — «⊥», приведение посредством правила (2):

входной символ — *IDLIST*, сдвиг в состояние *5*:



входной символ «⊥», приведение посредством правила (1):

входной символ — *S*, поэтому *HALT* (*ОСТАНОВ*).
Разбор успешно завершен.

Заметим, что после сдвига входным символом всегда является следующий символ, а после приведения — символ, к которому только что привело действие.

5.4. ПОСТРОЕНИЕ LR-ТАБЛИЦЫ РАЗБОРА

В этом разделе мы покажем, как строятся таблицы LR-разбора на основании грамматики. Нам придется ссылаться на конкретную позицию в правиле, поэтому введем понятие конфигурации. Например, в (расширенной) грамматике

(1) *S*→ . **real** *IDLIST*
(2) *IDLIST*→*IDLIST*, *ID*
(3) *IDLIST*→*ID*
(4) *ID*→*A*|*B*|*C*|*D*

точка соответствует конфигурации (*1*, *0*), т. е. правилу (1) позиции 0; конфигурация (*1*, *1*) соответствовала бы точке, появляющейся сразу после **real** в правиле (1), а (*2*, *0*) — точке, появляющейся перед *IDLIST* в правой части правила (2). Конфигурации будут использоваться для представления продвижения в разборе. Так, конфигурация (*2*, *2*) покажет нам, что правая часть правила (2) распознана по запятую включительно. На любом этапе разбора может быть частично распознано некоторое число правых частей правил.

Состояния в таблице разбора примерно соответствуют конфигурациям в грамматике с той лишь разницей, что конфигурации, которые неразличимы для анализатора, представляются одним и тем же

состоянием. Например, если (*1,0*) соответствует состоянию *1*, а (*1, 1*) — состоянию *2*, то в вышеприведенной грамматике (*2, 0*), (*3, 0*) и (*4, 0*) будут также соответствовать состоянию *2*. Мы говорим, что множество конфигураций

$$\{(1, 1), (2, 0), (3, 0) (4, 0)\}$$

образуют замыкание (*1, 1*).

Из заданного состояния, не соответствующего концу правила, можно перейти в другое состояние, введя терминальный или нетерминальный символ. Это состояние называют преемником первоначального состояния. Чтобы построить таблицу разбора, необходимо прежде найти все состояния в грамматике. Поэтому мы начинаем с конфигурации (*1, 0*) и последовательно выполняем операции замыкания и образования преемника до тех пор, пока все конфигурации не окажутся включенными в какие-либо состояния. Там, где ряд конфигураций содержится в одном замыкании, каждая из них будет соответствовать одному и тому же состоянию. Новая конфигурация, которая получается при операции образования преемника, называется базовой. Если за базовой конфигурацией следует нетерминал, то все конфигурации, соответствующие помещению точки слева от каждой правой части правила для данного нетерминала (и т. д. рекурсивно), сконцентрируются в замыкании этой базовой конфигурации. Так, в предыдущей грамматике четко видны семь состояний, которые можно описать следующим образом:

	База	*Замыкание*
Состояние *1*	(*1, 0*)	{(*1, 0*)}
Состояние *2*	(*1, 1*)	{(*1, 1*), (*2, 0*), (*3, 0*), (*4, 0*)}
Состояние *3*	(*4, 1*)	{(*4, 1*)}
Состояние *4*	(*3, 1*)	{(*3, 1*)}
Состояние *5*	{(*2, 1*), (*1, 2*)}	{(*2, 1*), (*1, 2*)}
Состояние *6*	(*2, 2*)	{(*2, 2*), (*4, 0*)}
Состояние *7*	(*2, 3*)	{(*2, 3*)}

Эти состояния расположены в грамматике следующим образом:

(1) $S \rightarrow_1 \mathbf{real}_2 IDLIST_5$
(2) $IDLIST \rightarrow_2 IDLIST_{5,6}\, ID_7$
(3) $IDLIST \rightarrow_2 ID_4$
(4) $ID \rightarrow_{(2,6)} A|B|C|D_3$

Заметим, что конфигурация может соответствовать более чем одному состоянию, и в базе может быть более одной конфигурации, если преемники двух конфигураций в одном и том же замыкании неразличимы. Например, в вышеприведенном примере за конфигурациями (*1, 1*) и (*2, 0*) следует *IDLIST*, что делает (*1, 2*) и (*2, 1*) неразличимыми, *пока не* осуществится операция замыкания (которая в нашем случае не дает дополнительных конфигураций). Число состояний в анализаторе соответствует числу *множеств* неразличимых конфигураций в грамматике. Причина того, что два или более состояний соответствуют одной конфигурации, раскрывается в ходе разбора. Так, в каком-то состоянии может содержаться информация «левого контекста». Тем не менее в конце правила (4) появляется только одно состояние (состояние *3*), поскольку мы не допускаем более одного состояния для одного и того же *множества* конфигураций. Таким образом, со-

Таблица 5.2

Состояние	*S*	*IDLIST*	*ID*	real	«,»	*A* *B* *C* *D*	«⊥»
1				*S2*			
2		*S5*	*S4*			*S3*	
3							
4					*S6*		
5							
6			*S7*			*S3*	
7							

стояние *3* не содержит какой-либо информации левого контекста. Из рассмотренных выше множеств базы и замыкания вытекает, что состояния *2* и *6 не* соответствуют одному и тому же множеству конфигураций.

Действия анализатора со сдвигом аналогичны операциям получения преемника. Поэтому действия со сдвигом в таблицу разбора могут вноситься на основании информации о размещении состояний в грамматике (табл. 5.2). Позиция элементов таблицы вытекает непосредственно из вышеприведенной грамматики. Например, правило (2) означает: «из состояния *2* при чтении *IDLIST* перейти в состояние *5*», «из состояния *5* при чтении *запятой* перейти в состояние *6*» и т. д. Задача внесения в таблицу действий приведения довольно сложна. Осуществление приведения может в целом зависеть от текущего левого контекста и предварительно просматриваемого символа. Однако единственные состояния, в которых приведения возможны, — это состояния, соответствующие окончаниям правил (в нашем примере состояния *3, 4, 5* и *7*). В особом случае LR(0)-грамматики предварительно просматриваемый символ не имеет к этому отношения, и действия по приведению могут помещаться в каждый столбец таблицы в любом состоянии, соответствующем окончанию правила. Если бы данная грамматика была LR(0)-грамматикой, мы могли бы поместить *R*4 во все столбцы состояния *3*, *R3* — во все столбцы состояния H, R1 — во все столбцы состояния *5* и *R*2 — во все столбцы состояния *7*. Однако в состоянии *5* в одном столбце уже имеется элемент сдвига. Мы не можем помещать элемент приведения в ту же графо-клетку, и у нас возникает *конфликт сдвиг/приведение*. Алгоритм построения LR(0) не срабатывает (так как грамматика не обладает признаком LR(0)), и мы должны попробовать что-нибудь другое. Проблема здесь возникает лишь в связи с состоянием *5*, поэтому фактически все остальные элементы приведения мы можем внести и получить табл. 5.3.

Состояние *5* называется *неадекватным*. Попытаемся разрешить эту проблему, задавая предварительно просматриваемые символы, которые показали бы приведение в состоянии *5*, а не сдвиг. Из правил (1) и (2) видим, что такими символами могут быть только «⊥» и «,», а приведение возможно лишь в том случае, если символом окажется «⊥», в то время как анализатор осуществит сдвиг в состояние *6*, если следующим символом будет «,». Поэтому мы вносим *R*1 в пятую строку столбца, соответствующего знаку «⊥». Это не вызывает никакого конфликта, и неадекватность состояния снимается. В ситуациях когда

Таблица 5.3

Состояние	*S*	*IDLIST*	*ID*	**real**	«,»	*A B C D*	« ⊥ »
1				*S2*			
2		*S5*	*S4*				
3	*R4*	*R4*	*R4*	*R4*	*R4*	*R4*	*R4*
4	*R3*	*R3*	*R3*	*R3*	*R3*	*R3*	*R3*
5					*S6*		
6			*S7*			*S3*	
7	*R2*	*R2*	*R2*	*R2*	*R2*	*R2*	*R2*

все неадекватности можно разрешить таким образом, грамматику считают простой LR(1)- или SLR(1)-грамматикой.

Аналогичным образом, рассмотрев предварительно символы, можно исключить из таблицы несколько действий приведения в состояниях *3*, *4* и *7*. Если этого не сделать, то некоторые синтаксические ошибки будут обнаружены на более поздних шагах анализа (но не позднее в смысле считанного исходного текста). Исключение этих элементов и включение элемента *HALT* дает нам табл. 5.4, с которой мы уже встречались ранее.

Таблица 5.4

Состояние	*S*	*IDLIST*	*ID*	**real**	«,»	*A B C D*	« ⊥ »
1	*HALT*			*S2*			
2		*S5*	*S4*			*S3*	
3					*R4*		*R4*
4					*R3*		*R3*
5					*S6*		*R1*
6			*S7*			*S3*	
7					*R2*		*R2*

Включению элемента *HALT* способствует добавление к грамматике дополнительного правила

$$S' \rightarrow_i S_{HALT} \perp$$

Различие между LR(1)- и SLR(1)-грамматиками заключается в том, что при определении предварительно просматриваемых символов в алгоритме построения SLR(1) никакого внимания не уделяется левому контексту, тогда как в более общем случае левый контекст учитывается и даже играет решающую роль при решении вопроса о том, можно ли считать заданный символ действительно символом-следователем. Однако для устранения неадекватности в наиболее универсальных LR(1)-грамматиках иногда возникает необходимость ввести в таблицу разбора гораздо большее число состояний. Это осуществляется за счет переопределения конфигурации путем включения в нее множества символов-следователей, преемников, которые (для конкретного разбора) могут оказаться предварительно просматриваемыми символами в тех случаях, когда выполняется приведение в соответствие с порождающим правилом. Например, если вновь обратиться к грамматике, кото-

рая уже использовалась в данном разделе, то состояние *1* можно определить посредством

$$(1, 0), \perp$$

а состояние *6* — с помощью двух конфигураций:

$$(2, 2), (\langle\perp\rangle, \langle,\rangle)$$
$$(4, 0), (\langle\perp\rangle, \langle,\rangle)$$

Символы-следователи — это такие символы, которые правомочно могут следовать за нетерминалом в левой части правила при выполнении приведения. Когда же дело доходит до включения действий приведения, их только помещают в столбцы, содержащие действительные символы-следователи для конкретной конфигурации. В наиболее общем алгоритме построения LR состояния, соответствующие идентичным множествам конфигураций (в смысле SLR(1)), но имеющие неидентичные множества символов-следователей, считаются различными. Это может привести к значительному увеличению числа состояний в таблице разбора, а значит, и размера самой таблицы. Тем не менее этот алгоритм позволяет нам разбирать все языки, которые можно генерировать с помощью LR(1)-грамматик.

К счастью, полная универсальность от алгоритма построения LR(1) требуется редко, и на практике большинство языков программирования имеют свойства *SLR(1)*. Как видно из рис. 5.1, SLR(1)-грамматики включают грамматики со слабым предшествованием и с простым предшествованием, а также LL(1)-грамматики.

Даже в тех случаях, когда грамматика не обладает признаком SLR(1), полная LR(1)-обработка требуется не всегда. Если состояния, которые являются идентичными, за исключением множеств символов-следователей, сливаются в единое состояние, где различные следователи объединяются в единое множество, и если при организации

Рис. 5.1. Иерархия грамматик

таблицы разбора не возникают никакие неадекватности, то такую грамматику называют LALR(1)-грамматикой (LR(1) с предварительным просмотром символов). LALR(1)-таблица разбора имеет то же число состояний, что и SLR(1)-таблица, но множества символов-следователей в ней учитывают левый контекст.

Рассмотрим грамматику:

(1) $S \rightarrow_1 T_2 \textbf{else}_3 F_{4;11}$
(2) $T \rightarrow_1 E_5$
(3) $T \rightarrow_1 i_{6;8}$
(4) $F \rightarrow_3 E_7$
(5) $E \rightarrow_{(1,3)} E_{(5,7)} +_9 i_{10}$
(6) $E \rightarrow_{(1,3)} i_{(6,12)}$

где добавлены номера состояний.

Эти 12 состояний соответствуют следующим конфигурациям (в расширенном смысле этого термина):

состояние *1*	(*1, 0*),{«⊥»}
	(*2, 0*),{**else**}
	(*3, 0*),{**else**}
	(*5, 0*),{**else**«+»}
	(*6, 0*),{**else**«+»}
состояние *2*	(*1, 1*),{«⊥»}
состояние *3*	(*1, 2*),{«⊥»}
	(*4, 0*),{«;»}
	(*5, 0*),{«;»,«+»}
	(*6, 0*),{«;»,«+»}
состояние *4*	(*1, 3*),{«⊥»}
состояние *5*	(*2, 1*),{**else**}
	(*5, 1*),{**else**,«+»}
состояние *6*	(*3, 1*),{**else**}
	(*6, 1*),{**else**,«+»}
состояние *7*	(*4, 1*),{«;»}
	(*5, 1*),{«;»,«+»}
состояние *8*	(*3, 2*),{**else**}
состояние *9*	(*5, 2*),{«;»,**else**,«+»}
состояние *10*	(*5, 3*),{«;»,**else**,«+»}
состояние *11*	(*1, 4*),{«⊥»}
состояние *12*	(*6, 1*),{«;»,«+»}

При определении множеств символов-следователей применяется ряд правил:

1. В конфигурации (*1, 0*) множество символов-следователей состоит из {⊥}, так как левой частью порождающего правила служит начальный символ, не появляющийся в правой части ни одного из правил. (Для грамматики, в которой начальный символ не обладает этим свойством, желательно ввести с помощью дополнительного правила новый начальный символ S', обладающий указанным свойством, $S' \rightarrow S \perp$.)

2. При выполнении операции замыкания с целью определения других конфигураций, соответствующих какому-либо базовому состоянию (состояниям), можно также определять множества символов-следователей. Например, поскольку конфигурация (*1,0*) предшествует нетерминалу T, конфигурации, соответствующие левым краям правых частей порождающих правил для T, находятся в замыкании, и символы-следователи T в *порождающем правиле* (1) составляют множества символов-следователей. Таким образом,

(*2, 0*), {**else**}
(*3, 0*), {**else**}

представляют собой конфигурации (в смысле LALR(1)) состояния *1*. *(5,0)* и *(6,0)* также являются конфигурациями в состоянии 1. В случае (6,0) **else** находится в множестве символов-следователей, так как может следовать за *E*, поэтому есть символ-следователь T. Символ + также находится в множестве символов-следователей, поскольку *E* можно «вызвать» из правила «5». Итак, конфигурация LALR(1) имеет вид

$$(6, 0), \{\text{else}, «+»\}$$

Аналогично

$$(5, 0), \{\text{else}, «+»\}$$

находится в состоянии *1*.

3. Выполнение операции нахождения символа-следователя для конфигурации не влияет на множество символов-следователей. Например, *(1, 0)* в состоянии *1* и *(1, 1)* в состоянии *2* имеют одно и то же множество символов-следователей ($\{\perp\}$). Однако в состоянии *9* (конфигурация *(5, 2)*) множества символов-следователей, соответствующие конфигурациям *(5, 1)*, {**else**, «+»} (состояние *5*) и *(5, 1)*, {«;», «+»} (состояние *7*), слиты вместе, так как алгоритм LALR не допускает разных состояний для неразличимых множеств SLR (или LR(0)) конфигураций. Таблица разбора с включенными элементами сдвигов представлена табл 5.5.

Таблица 5.5

Состояние	*S*	*T*	*F*	*E*	**else**	«;»	«+»	*i*	«$\perp$»
1	*S2*			*S5*				*S6*	
2					*S3*				
3			*S4*	*S7*				*S12*	
4						*S11*			
5							*S9*		
6						*S8*			
7							*S9*		
8									
9								*S10*	
10									
11									
12									

Эта грамматика не обладает признаком LR(0), так как, например, мы не можем поместить элементы приведения в каждый столбец для состояния *5*, что может соответствовать окончанию правила. Не обладает она также и признаком SLR(1), так как в противном случае из правила (6) мы могли бы взять элемент приведения в столбец «;» для состояния *6*, поскольку «;» есть действительный символ-следователь *E* (раз он следует за *F* в смысле SLR(1)). Однако включение действия приведения в столбец «;» состояния 6 приведет к конфликту с уже имеющимся в этой позиции действием сдвига. Конфликт можно разрешить, если, как видно из дополненных конфигураций, соответствующих каждому состоянию, **else** и «+» служат единственными действительными символами-следователями в состоянии *6*. Таким образом, алгоритм LALR только поместит действия *R*6 в столбцы **else** и «+» в состоянии *6*. Продолжив этот процесс далее, мы построим полную таблицу LALR(1)-разбора (табл. 5.6). Здесь относительно не-

Таблица 5.6

Состояние	*S*	*T*	*F*	*E*	else	«;»	«+»	*i*	«⊥»
1	*HALT*	*S2*		*S5*				*S6*	
2					*S3*				
3			*S4*	*S7*				*S12*	
4						*S11*			
5					*R2*		*S9*		
6					*R6*	*S8*	*R6*		
7						*R4*	*S9*		
8					*R3*				
9								*S10*	
10					*R5*	*R5*	*R5*		
11									*R1*
12						*R6*	*R6*		

много элементов приведения, и отсутствуют конфликты. Если бы алгоритм LALR(1) не смог разрешить какие-либо конфликты, на следующем шаге мы должны были бы попытаться применить алгоритм LR(1). Для этого нам понадобилось бы ввести дополнительные состояния. Например, состояние *10* пришлось бы разбить на два состояния, соответствующие

$$(5, 3), \{«;», «+»\}$$

и

$$(5, 3), \{\textbf{else}, «+»\}$$

причем каждое состояние соответствует одной и той же конфигурации (конфигурациям), но имеет различные символы-следователи. Таким образом, будет запоминаться больший объем левого контекста — в данном случае: должно ли E приводиться к F или T. Анализатору фактически не нужна эта информация, так как состояния, в которых E приводится к T и к F, различны. Потребность в алгоритме LR(1) возникает редко, но когда она возникает, число используемых состояний обычно намного превышает то, которое бывает в алгоритме LALR(1).

Описанный ранее алгоритм разбора остается постоянным независимо от того, какая разбирается грамматика: LR(0), SLR(1), LALR(1) или LR(1). Только алгоритм построения таблицы разбора становится сложнее по мере возрастания универсальности грамматики. Размер таблицы, т. е. число состояний, одинаков вне зависимости от используемого алгоритма — LR(0), SLR(1) или LALR(1). Лишь LR(1)-алгоритм создает бóльшую таблицу. Метод, позволяющий решить, обладает ли грамматика признаком LR(1), заключается в попытке сформировать таблицу LR(0)-разбора. Если это можно осуществить без возникновения каких-либо конфликтов, грамматика оказывается LR(0)-грамматикой, а следовательно, и LR(1)-грамматикой. Если же конфликты возникают, опробуется SLR(1)-алгоритм. При успешной попытке грамматика будет SLR(1)-грамматикой, а следовательно, и LR(1)-грамматикой. В ином случае пробуется алгоритм LALR(1), и если это разрешает все конфликты, то можно считать, что грамматика обладает признаком LALR(1), а значит, и признаком LR(1). И наконец, если конфликты остаются, выполняется алгоритм LR(1), и грамматика является LR(1)-грамматикой или нет в

зависимости от того, какие именно конфликты имеют место: сдвиг/приведение или приведение/приведение. Конечно, можно было бы сразу опробовать алгоритм LR(1), но в подавляющем большинстве случаев на это ушло бы гораздо больше времени, чем на опробование сначала более простых алгоритмов. Типичный генератор LR(1)-анализаторов действует, опробуя различные алгоритмы в порядке возрастания их сложности. Грамматики языков программирования часто имеют признак SLR(1) и почти всегда признак LALR(1).

Отдельный класс грамматик, не обладающих признаком LR(1), составляют неоднозначные грамматики (см. разд. 2.4). Задача определения, однозначна ли грамматика или нет, неразрешима. Тем не менее, поскольку ни одна из LR(1)-грамматик не является неоднозначной, LR(1)-алгоритм может показать нам, что данная грамматика однозначна. Однако отсюда *не* следует, что отказ LR(1)-алгоритма свидетельствует о неоднозначности грамматики. Иначе мы могли бы проверить неоднозначность грамматики, что противоречило бы неразрешимости проблемы.

Рассмотренные выше прямоугольные таблицы разбора позволяют осуществлять быструю выборку и обеспечивают широкие диагностические возможности (так как каждый элемент — пробел можно ассоциировать с различными сообщениями об ошибках). Однако эти таблицы занимают слишком большой объем памяти (до 50 К слов для синтаксиса Алгола 68). Существуют хорошо известные методы записи в память неплотных матриц, и многие из них можно применить к LR-таблицам разбора. Кроме того, есть и другие эффективные методы хранения таблиц, которые учитывают их конкретные свойства [28]. Однако, как и следовало ожидать, чем-то приходится поступиться ради более эффективного использования объема памяти. Это может быть время обращения к определенному элементу таблицы разбора или даже более позднее обнаружение синтаксических ошибок (в том смысле, что анализатор успеет выполнить больше действий, прежде чем обнаружится ошибка, а не в том, что будет считано большее число символов программы).

5.5 СРАВНЕНИЕ LL- и LR-МЕТОДОВ РАЗБОРА

Как LL-, так и LR-методы разбора имеют много достоинств. Оба они — детерминированные и могут обнаруживать синтаксические ошибки, как правило, на самом раннем этапе. LR-методы обладают тем преимуществом, что они применимы к более широкому классу грамматик и языков и преобразования грамматик в них обычно не требуются. Однако стоит посмотреть, какими оказываются эти теоретические преимущества LR-анализаторов на практике.

Если кто-либо для преобразования контекстно-свободной грамматики для Алгола 68 по пересмотренному сообщению воспользуется хорошим преобразователем грамматики, то из опыта можем сказать, что только несколько частей грамматики окажутся трудными для преобразования и их придется преобразовывать вручную.

Приведем три примера тех средств грамматики, которые вызвали у нас затруднения (подробнее об этом см. в [31]).

1. Допустим, в грамматике G_1 есть правила:

S→CO|CL
CO→orb EC stick SC stick SC crb

```
EC→u semi EC|u
CL→orb SC crb
SC→PLU semi SC|PLU
PLU→lab PLU|u
```

{*CO* — условное предложение, *CL* — замкнутое предложение, *EC* — выясняющее предложение, *SC* — последовательное предложение, *PLU* — «возможно помеченный элемент», *u* — элемент, *orb* — открывающая круглая скобка и т. д.}

Проблема возникает в связи с тем, что элементы не могут иметь меток в выясняющем предложении (появляющемся в условном предложении), но могут их иметь в последовательном предложении (появляющемся в замкнутом предложении), а преобразователь грамматики с этим не может справиться. Данный *язык* тем не менее обладает признаком LL(1), но нужно «слить воедино» правила для замкнутых и условных предложений, чтобы получить LL(1)-грамматику G_2 с правилами

```
S→orb T
T→lab V|uW
V→lab V|uX
W→crb|semi T|stick SC stick SC crb
X→semi V|crb
SC→PLU Y
Y→semi PLU Y|ε
PLU→lab PLU|u
```

Грамматика G_2 будет LL(1)-грамматикой.

2. Предположим, что грамматика G_3, которая, если придать ей несколько обобщенный характер, генерирует фактические описатели структур на Алголе 68, имеет правила:

```
STRAD→struct FPACK
FPACK→orb FIELDS crb
FIELDS→FIELD comma FIELDS|FIELD
FIELD→ad LIST
LIST→tag|tag comma LIST
```

Здесь опять наш преобразователь грамматики не может выдать LL(1)-грамматику. Проблема заключается в двойном использовании запятой, а грамматика аналогична приведенной в начале разд. 5.2 (а также в разд. 4.4). Преобразование вручную, подобное вышеописанному, дает LL(1)-грамматику.

3. Другая часть грамматики Алгола 68, которая трудна для преобразования в LL(1)-форму, касается форматов. Например, включение определяется следующим образом (грамматика G_4):

```
INSERT→LIT ALIGNS|ALIGNS|LIT
LIT→REPL denote LIT1|REPL denote
REPL→nerepl|ε
LIT1→nerepl denote|nerepl denote LIT1
ALIGNS→ALIGN ALIGNS|ALIGN
ALIGN→REPL code LIT|REPL code
```

{*INSERT* — вставка, *ALIGN* — размещение, *LIT* — литерал, *REPL* — повторитель, *nerepl* — непустой повторитель, *denote* — обозначение и т. д.} G_4 можно вручную преобразовать в LL(1)-форму (G_5):

```
INSERT→nerepl ANER|denote ADEN|code ACODE
ANER→denote ADEN|code ACODE
ADEN→nerepl ANER|code ACODE|ε
ACODE→INSERT|ε
```

Три грамматики G_1, G_3 и G_4 представляют три принципиальных случая, где преобразователь грамматик не смог преобразовать нашу грамматику Алгола 68 в LL(1)-форму. Интересно поэтому выяснить, являются ли эти грамматики LR(1)-грамматиками.

Грамматика G_1 не обладает признаком LR(1), так как при чтении строки

(u

с *semi* в качестве предварительно просматриваемого символа не известно, выполнять ли приведение к *PLU* с помощью второго варианта для *PLU* или сдвиг в соответствии с первым вариантом для *EC*. Так что здесь имеет место неразрешимый конфликт сдвиг/приведение.

Грамматика G_3 не обладает признаком LR(1), потому что при чтении строки

struct (ad tag

с предварительно просматриваемым символом *comma* не известно, выполнять ли приведение к *LIST*, как в первом варианте для *LIST*, или сдвиг, как во втором варианте. Опять мы имеем дело с неразрешимым конфликтом сдвиг/приведение.

Грамматика G_4 также не обладает признаком LR(1), поскольку при чтении строки

code

с предварительно просматриваемым символом *nerepl* не известно, выполнять ли приведение с помощью второго варианта для *ALIGN* или сдвиг в соответствии с первым вариантом. Мы вновь сталкиваемся с конфликтом сдвиг/приведение.

Поэтому в тех случаях, когда требовались преобразования вручную, чтобы привести грамматику к LL(1)-форме, эту грамматику все же требовалось преобразовать, прежде чем она смогла бы послужить основой для LR-анализатора. Таким образом, теоретическое преимущество LR-анализаторов перед LL-анализаторами (т. е. их бóльшая универсальность) оказывается не таким уж значительным на практике. При наличии хорошего преобразователя грамматики (за исключением вышеупомянутых проблем) сам факт необходимости преобразовать грамматику не вызывает в действительности у разработчика компилятора никаких затруднений, и в большинстве случаев ему даже не приходится рассматривать эту преобразованную грамматику. Таких случаев, как мы видели ранее, когда преобразователь оказывается несостоятельным и либо зацикливается, либо выдает неправильные диагностические сообщения, относительно немного. К сожалению, эти части грамматики часто не обладают признаком LR(1), и трудности возникают независимо от того, какой метод разбора применяется.

Существуют серьезные причины, по которым мы хотим избежать преобразований вручную, если это возможно: во-первых, трудно выбрать вид преобразования, во-вторых, мы не всегда можем гарантировать, что они не изменят генерируемый язык.

Эти два метода разбора можно также сравнивать в отношении размеров таблиц и времени разбора. Использование по одному слову на элемент таблицы LL-разбора позволяет свести размер типичной таблицы разбора для Алгола 68 примерно к 4К слов. Соответствующий размер таблицы LR-разбора, описанный в разд. 5.3, составляет при-

мерно 50 К слов. Однако такое сравнение не совсем справедливо, поскольку применение методов оптимизации, упоминаемых в последнем разделе, дает возможность сократить эту цифру процентов на 90. Так что LL-метод в отношении размера таблицы разбора не представляется лучшим, чем LR-метод.

Коэн и Рот [14] сравнили максимальное, минимальное и среднее время разбора предложений с помощью LL- и LR-анализаторов. Они приводят показатели относительной эффективности этих методов при их использовании на машине DEC PDP-10. Результаты показывают, что LL-метод «быстрее» процентов на 50.

Оба метода разбора позволяют включить в синтаксис действия для выполнения некоторых аспектов процесса компиляции. В следующей главе мы увидим, как это осуществляется в LL-анализаторах. Для LR-анализаторов такие действия обычно связаны с приведениями, и часто в грамматику там, где требуется не окончание правила, а другие действия, вводятся фиктивные правила. Это не влияет на признак принадлежности грамматики к LR(1) до тех пор, пока грамматика обладает LL(1)-свойством.

Разные составители компиляторов отдают предпочтение разным методам (т. е. разбору снизу вверх или сверху вниз), что часто служит предметом дискуссии. На практике выбор метода в основном зависит от наличия хорошего генератора синтаксических анализаторов любого типа.

Упражнения

5.1. Приведите грамматики, отличные от описанных в настоящей главе, которые обладают признаком

а) LR(0);
б) SLR(1), но не LR(0);
в) LR(1), но не SLR(1).

5.2. Определите, какие из следующих грамматик являются LR(1)-грамматиками и обоснуйте свой вывод:

а) $E \to E + E$
$\quad E \to i$
б) $Z \to 0S0$
$\quad \to 1S1$
$\quad \to 0$
$\quad \to 1$
в) $S \to 0S0$
$\quad \to 0S1$
$\quad \to c$

Там, где грамматика не обладает признаком LR(1), приведите эквивалентную LR(1)-грамматику, если такая существует.

5.3. Постройте таблицу разбора для грамматики, имеющей следующие правила (S — начальный символ):

$S \to E$ $\quad$ $T \to F$
$E \to E + T$ $\quad$ $F \to (E)$
$E \to T$ $\quad$ $F \to x$
$T \to T \times F$ $\quad$ $F \to y$

Является ли эта грамматика SLR(1)-грамматикой?

5.4. Докажите, что следующая грамматика не обладает признаком LR(1) (PROGRAM — начальный символ):

$$PROGRAM \rightarrow begin\ DECLIST\ semi\ STATELIST\ end$$

$$DECLIST \rightarrow \underset{d}{d}\ semi\ DECLIST$$

$$STATELIST \rightarrow \underset{s}{s}\ semi\ STATELIST$$

Преобразуйте эту грамматику в эквивалентную SLR(1)-грамматику и постройте соответствующую таблицу разбора.

5.5. Постройте таблицу разбора для грамматики (S — начальный символ):

$$S \rightarrow a\ t\ F$$
$$F \rightarrow \underset{\varepsilon}{,}\ ITEM\ F$$
$$ITEM \rightarrow \underset{t}{a}\ t$$

5.6. Докажите, что следующая грамматика не обладает признаком LR(1) (S — начальный символ):

$$S \rightarrow 1S0$$
$$S \rightarrow 0S1$$
$$S \rightarrow 1\ 0$$
$$S \rightarrow 0\ 1$$

5.7. Обладает ли а) грамматика из упражнения 5.6 и б) язык, генерируемый этой грамматикой, LR(k)-свойством для любого k?

5.8. Покажите, что следующая грамматика является не LR(1)-грамматикой, а LR(2)-грамматикой:

$$S \rightarrow V := E$$
$$S \rightarrow L\ S$$
$$L \rightarrow I:$$
$$V \rightarrow I$$

Предложите, что можно сделать на стадии лексического анализа в отношении непринадлежности ее к LR(1)?

5.9. Какие преимущества могло бы иметь построение SLR(1)-таблицы разбора на основании LR(0)-грамматики?

5.10. Требуется ли для метода LR-разбора использование двух стеков (стека символов и стека состояний) или эти два стека можно объединить в один? Обоснуйте свой ответ.

5.11. Докажите, что все регулярные языки обладают LR(1)-свойством.

Глава 6

ВКЛЮЧЕНИЕ ДЕЙСТВИЙ В СИНТАКСИС

Анализ и синтез в компиляции, хотя их часто удобно рассматривать как отдельные процессы, во многих случаях происходят параллельно. Это совершенно очевидно для однопроходного компилятора, но и в многопроходных компиляторах генерирование объектного или какого-либо промежуточного кода большей частью осуществляется одновременно с синтаксическим анализом. Последнее не должно вызывать удивления: ведь как только синтаксический анализатор распознает, например, присваивание, он, естественно, выдаст код для присваивания в данной точке. В настоящей главе мы покажем, что в грамматике могут содержаться вызовы действий не только для генерирования кода, но и для выполнения других задач компиляции, таких, как построение таблиц символов и обращение к ним и т. п.

6.1. ПОЛУЧЕНИЕ ЧЕТВЕРОК

В качестве примера включения действия в грамматику для генерирования кода рассмотрим проблему разложения арифметических выражений на четверки. Выражения определяются грамматикой со следующими правилами:

$$
\begin{aligned}
S &\rightarrow EXP \\
EXP &\rightarrow TERM \\
&\quad\ EXP + TERM \\
TERM &\rightarrow FACT \\
&\quad\ TERM \times FACT \\
FACT &\rightarrow -FACT \\
&\quad\ ID \\
&\quad\ (EXP) \\
ID &\rightarrow a \mid b \mid c \mid d \mid e
\end{aligned}
$$

где S — начальный символ.

Примеры выражений:

$$
\begin{aligned}
&(a+b)\times c \\
&a\times b+c \\
&a\times b+c\times d\times e
\end{aligned}
$$

Все идентификаторы содержат одну букву: это исключает необходимость выполнять лексический анализ.

Грамматика для четверок имеет следующие правила:

$$
\begin{aligned}
QUAD &\rightarrow OPERAND\ OP1\ OPERAND = INT \\
&\quad\ OP2\ OPERAND = INT \\
OPERAND &\rightarrow INT \\
&\quad\ ID \\
INT &\rightarrow DIGIT \\
&\quad\ DIGIT\ INT
\end{aligned}
$$

$$DIGIT \rightarrow 0|1|2|3|4|5|6|7|8|9$$
$$ID \rightarrow a|b|c|d|e$$
$$OP1 \rightarrow +|\times$$
$$OP2 \rightarrow -$$

Примеры четверок:

$$-a=4$$
$$a+b=7$$
$$6+3=11$$

Выражение

$$(-a+b)\times(c+d)$$

будет соответствовать такой последовательности четверок:

$$-a=4$$
$$1+b=2$$
$$c+d=3$$
$$2\times 3=4$$

Целые числа с левой стороны от знаков равенства относятся к другим четверкам. Из сформированных четверок нетрудно генерировать машинный код, а многие компиляторы на основании четверок осуществляют трансляцию в промежуточный код. Цель настоящего раздела — показать, как включать в грамматику выражений действия для генерирования соответствующих четверок. Действия заключаются в угловые скобки <,> и обозначаются как *A1, A2...* . В данном случае требуются четыре различных действия. Алгоритм пользуется стеком, а номера четверок размещаются с помощью целочисленной переменной. Перечислим эти действия:

A1 — поместить элемент в стек;

A2 — взять три элемента из стека, напечатать их с последующим знаком «=» и номером следующей размещаемой четверки и поместить полученное целое число в стек;

A3 — взять два элемента из стека, напечатать их с последующим знаком «=» и номером следующей размещаемой четверки и поместить полученное целое число в стек;

A4 — взять один элемент из стека.

Грамматика с учетом этих добавленных действий примет вид

$$S \rightarrow EXP<A4>$$
$$EXP \rightarrow TERM$$
$$EXP+<A1>TERM<A2>$$
$$TERM \rightarrow FACT$$
$$TERM\times<A1>FACT<A2>$$
$$FACT \rightarrow -<A1>FACT<A3>$$
$$ID<A1>$$
$$(EXP)$$
$$ID \rightarrow a|b|c|d|e$$

Действие *A1* используется для помещения в стек всех идентификаторов и операторов, а действия *A2* и *A3* — для получения бинарных и унарных четверок соответственно. Можно написать код для этих действий на Алголе 68, допуская следующие описания:

```
[1 : 20] union (int, char) stack;
int ptr := 0, quadno := 0;
char in
```

Считается, что *in* принимает значение последней считанной литеры (или символа).

Покажем действия *A1-A4*:

```
⟨A1⟩
stack [ptr plusab 1] := in

⟨A2⟩
begin
for i from ptr – 2 to ptr
do print stack [i]
od;
print (("=", quadno plusab 1, newline));
stack [ptr minusab 2] := quadno
end

⟨A3⟩
begin
for i from ptr – 1 to ptr
do print stack [i]
od;
print (("=", quadno plusab 1, newline));
stack [ptr minusab 1] := quadno
end
⟨A4⟩
ptr minusab 1
```

В качестве примера проследим за преобразованием выражения

$$(-a+b)\times(c+d)$$

в четверки. Действие *A1* выполняется после распознавания каждого идентификатора и оператора, действие *A2* — после второго операнда каждого знака бинарной операции, а действие *A3* — после первого (и единственного) операнда каждого знака унарной операции. Действие же *A4* выполняется только один раз после считывания всего выражения.

Последняя считанная литера	Действие	Выход
(	—	
—(минус)	*A1*, поместить в стек «—»	
a	*A1*, поместить в стек *a*	
	A3, удалить из стека *2*	$-a=1$
	Элементы, поместить в стек «*1*»	
+	*A1*, поместить в стек «+»	
b	*A1*, поместить в стек *b*	
	A2, удалить из стека *3*	$1+b=2$
	Элементы, поместить в стек «*2*»	
)		
×	*A1*, поместить в стек «×»	
(		
c	*A1*, поместить в стек *c*	
+	*A1*, поместить в стек «+»	
d	*A1*, поместить в стек *d*	
	A2, удалить из стека *3*	$c+d=3$
	Элементы, поместить в стек «*3*»	
)	*A2*, удалить из стека *3*	$2\times3=4$
	Элементы, поместить в стек «*4*»	
	A4, удалить из стека *1*	
	Элемент	

Следует заметить, что:

1) в рассмотренном примере нам ни разу не пришлось сравнивать приоритеты двух знаков операций, поскольку эта информация содержалась в грамматике;

2) этот метод нетрудно распространить на языки с множеством различных приоритетов знаков операций, например Алгол 68;

3) приведенная выше грамматика рекурсивна влево, так как правила вида

$$TERM \rightarrow FACT \times TERM$$

подразумевают иной порядок вычисления: например, $(A \times (B \times C))$, а не $((A \times B) \times C)$. Тем не менее мы можем преобразовать леворекурсивные правила в праворекурсивные с помощью включения действий в соответствующие места путем введения новых нетерминалов. Так, правила

$$TERM \rightarrow FACT$$
$$TERM \rightarrow TERM \times <A1> FACT <A2>$$

эквивалентны правилам

$$TERM \rightarrow FACT\ NEW$$
$$NEW \rightarrow \varepsilon$$
$$\times <A1> FACT <A2> NEW$$

Более того, это преобразование можно выполнять автоматически, поэтому для простоты лучше придерживаться терминалов первоначальных (леворекурсивных) правил.

Идею о генераторе синтаксического (LL)-анализатора нетрудно расширить включением действий в синтаксис. Точно так же, как «вызывается» правило из другого правила, можно «вызвать» действие, когда оно включено в грамматику.

Объем «кода» в вышеприведенном примере весьма невелик и, как мы увидим из других примеров, иллюстрирующих тот же метод, труднее всего определить, где в грамматике должны появляться действия, а не писать код для этих действий.

6.2. РАБОТА С ТАБЛИЦЕЙ СИМВОЛОВ

Поскольку синтаксический анализатор обычно использует контекстно-свободную грамматику, необходимо найти метод определения контекстно-зависимых частей языка. Например, во многих языках идентификаторы не могут применяться, если они не описаны, и имеются ограничения в отношении способов употребления в программе значений различных типов. Для запоминания описанных идентификаторов и их типов большинство компиляторов пользуется таблицей символов. В Алголе 68 таблица символов требуется также для записи информации относительно специфицируемых пользователем видов и обозначений операций.

Когда описывается идентификатор, например

```
int a
```

мы говорим, что у нас имеется *определяющая реализация a*. Однако *a* может встречаться и в другом контексте:

$$a := 4 \quad \text{или} \quad a + b \quad \text{или} \quad read\ (a)$$

и здесь *a* будут *прикладными реализациями*.

Если мы имеем дело с определяющей реализацией идентификатора (или специфицируемым пользователем видом либо знаком операции), то компилятор помещает объект в таблицу символов, а если с прикладной реализацией, — то в таблице символов осуществляется поиск элемента, соответствующего определяющей реализации объекта, чтобы узнать его тип и (возможно) другие признаки, требующиеся во время компиляции.

Во многих языках один и тот же идентификатор может использоваться для представления в разных частях программы различных объектов. В таких случаях структура программы помогает различать эти объекты, например

```
begin int a;
    .
    .
    .
end;
begin char a
    .
    .
    .
end
```

В первом блоке *a* имеет вид **ref int** в Алголе 68 (т. е. ее значение относится к объекту вида **int**), а во втором — **ref char**. Так, в этих двух блоках *a* представляет два *разных* объекта.

Таблица символов должна иметь ту же блочную структуру, что и программа, чтобы различать виды употребления одного и того же идентификатора.

Рассмотрим язык, обладающий следующими свойствами:

1. Определяющая реализация идентификатора появляется (текстуально) раньше любой прикладной реализации.

2. Все описания в блоке помещаются непосредственно за **begin**, т. е. раньше всех операторов или предложений.

3. При наличии прикладной реализации идентификатора соответствующая определяющая реализация находится в наименьшем включающем блоке, в котором содержится описание этого идентификатора.

4. В одном и том же блоке идентификатор не может описываться более одного раза.

Свойства 3 и 4 присущи практически всем языкам с блочной структурой, свойство 2 характерно для Алгола 60, а свойство 1 представляет собой ограничение, имеющееся в некоторых реализациях этого языка.

Допустим, что синтаксис описаний идентификаторов задается правилами:

DEC→**real** *IDS*|**integer** *IDS*|**boolean** *IDS*
IDS→*id*
IDS→*IDS*, *id*

а блок определяется как

BLOCK→**begin** *DECS*; *STATS* **end**

где

DECS→*DECS*; *DEC*
DECS→*DEC*
STATS→*STATS*; *s*
STATS→*s*

Таблица символов может иметь такую структуру, как показано на

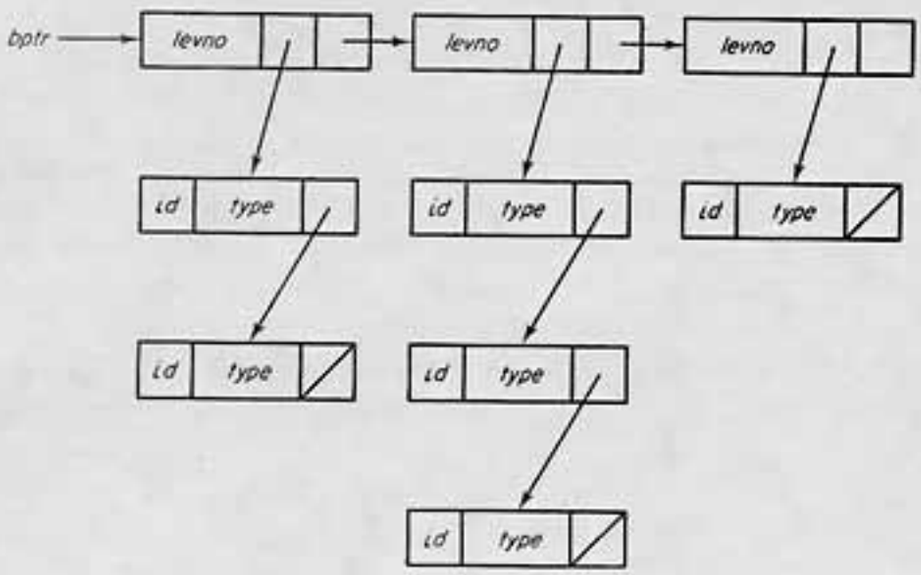


Рис. 6.1

В любой точке разбора в цепи находятся те блоки, в которые делается текущее вхождение, а уже описанные идентификаторы помещаются в список идентификаторов для того блока, где они описаны.

Для представления таблицы символов могут использоваться следующие структуры:

mode btab = struct (int *levno*, **ref idlist** *idl*, **ref btab** *next*)
mode idlist = struct (int *id*, *type*, **ref idlist** *next id*)

В этих структурах мы допускаем, что идентификаторы и типы представляются целыми числами. Имеется указатель на элемент таблицы символов, соответствующий наименьшему включающему блоку. Первоначально он описывается как

ref btab *bptr* := **nil** (нулевой указатель)

и выполняются действия, связанные с входом в блок и выходом из него:

BLOCK→**begin** <A1> *DECS; STATS*<*A2*>**end**

где

<*A1*>
bptr := **heap btab** := (*ln* **plusab** *1*, **nil**, *bptr*)

Здесь *ln* представляет номер уровня текущего блока (т. е. глубину вложения) и первоначально является нулем.

<*A2*>
begin *ln* **minusab** *1*;
bptr := *next* **of** *bptr*
end

Предусмотрено также действие для каждой определяющей реализации идентификатора. Это действие появляется в грамматике следующим образом:

IDS→*id*<*A3*>
IDS, *id*<*A3*>

(Заметим, что левая рекурсия позволит нам специфицировать особое действие, связанное с первым идентификатором в списке, а правая рекурсия — конкретное действие для последнего идентификатора.)

Это действие таково:

```
⟨A3⟩
begin ref idlist pnil = nil;
  ref idlist ptr := idl of bptr;
  bool error := false;
  while ptr isnt pnil
  do
    if id of ptr = iden then
    error := true; ptr := pnil
    fi;
    ptr := next id of ptr
  od;
  if error then print (("identifier", iden,
    "already declared"))
  else idl of bptr := heap idlist := (iden,
    type, idl of bptr)
  fi
end
```

iden — имя идентификатора, *type* — его тип. Здесь проверяется, был ли идентификатор уже описан в блоке или нет.

С прикладной реализацией идентификатора связано действие *A4*, которое отыскивает в таблице символов тип идентификатора:

```
⟨A4⟩
begin bool found := false;
  ref idlist pnil = nil;
  ref btab bnil = nil;
  ref btab b := bptr;
  while b isnt bnil and not found
  do ref idlist ptr := idl of b;
    while ptr isnt pnil and not found
    do if id of ptr = iden
      then type := type of ptr;
        found := true
      fi;
      ptr := next id of ptr
    od;
    b := next of b
  od;
  if not found
  then print ("identifier not declared");
    idl of bptr := heap idlist :=
    (iden, default type, idl of bptr);
    type := default type
  fi
end
```

Если идентификатор не был описан, он включается в таблицу символов, и его тип получает соответствующее значение по умолчанию. При этом исключается поток сообщений об ошибках.

На этом мы завершаем рассмотрение операций, производимых с таблицей символов, которые требуются для нашего простого языка.

В таблице символов настоящего компилятора может содержаться другая информация об идентификаторе, необходимая во время компиляции, например его адрес в ходе прогона или в случае константы — ее значение. Позднее мы покажем, как обобщаются эти алгоритмы для обращения с более сложными языками, например Алголом 68. Однако прежде чем перейти к другой теме, нам хотелось бы заметить, что существуют более простые алгоритмы ведения таблицы символов в языках, подобных рассмотренному в настоящем разделе. Но у этих простых алгоритмов есть недостаток — их трудно обобщать для работы со сложными языками.

В языке, обладающем перечисленными выше свойствами, в качестве структуры данных для таблицы символов очень удобен стек, каждым элементом которого служит элемент этой таблицы символов.

При встрече с описанием соответствующий элемент таблицы символов помещается в верхнюю часть стека, а при выходе из блока все элементы таблицы символов, соответствующие описаниям в этом блоке, удаляются из стека. Указатель же стека понижается до положения, которое он имеет при вхождении в блок. В результате в любой момент разбора элементы таблицы символов, соответствующие всем текущим идентификаторам, находятся в стеке, а связанные с ними прикладные и определяющие реализации идентификаторов требуют поиска в стеке в направлении сверху вниз. Применение стека вместо более сложной цепной структуры дает возможность сэкономить место, занимаемое указателями в этой цепной структуре.

Рассмотренный метод иллюстрируется следующим образом:

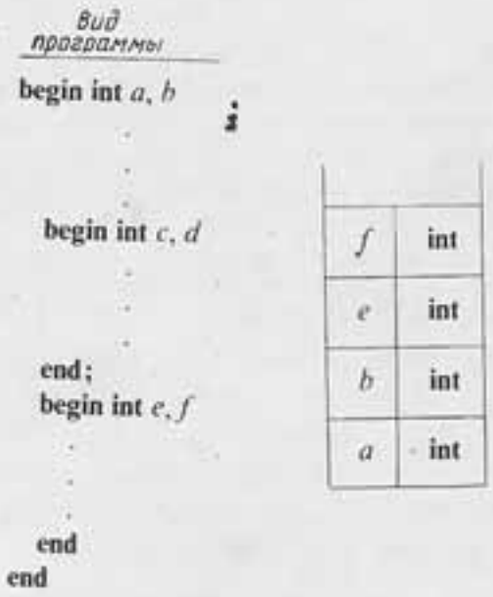


Таблица символов показана после встречи с **begin int** *e* и *f*.

6.3. ДРУГИЕ ПРИЛОЖЕНИЯ

Из приведенных выше примеров вытекает, что принцип включения действий в грамматику позволяет получить простой и элегантный компилятор, а действия выполняются на соответствующем уровне в грамматике. Составитель компилятора должен иметь возможность рассматривать каждый уровень отдельно, а грамматика обеспечивает рамки для компилятора в целом.

Как уже отмечалось ранее, компилятор — это просто программа, которая принимает на входе программу, написанную на определенном языке, и выдает на выходе программу в машинном коде. Выход в виде машинного кода не совсем уместен по отношению к синтаксическому анализатору, поэтому заслуживает внимания идея создания «синтаксического анализатора» с выходом какого-либо иного типа. Метод включения действий в грамматику можно расширить так, чтобы выдавать почти любую программу, ввод которой определяется посредством грамматики. Покажем это на примерах.

1. *Программа резмещения*

Мысль о создании программы, которая принимает в качестве входа любую программу на языке типа Алгол 60 или Алгол 68 и выдает программу, размещенную в точном соответствии с каким-либо условием, не нова. Таких программ написано много. Один из методов их создания заключается в применении к выбранному языку синтаксического анализатора, дополненного подходящими для выполнения размещения действиями. Помимо прочего, условия размещения почти наверняка тесно связаны с синтаксической структурой программы, и поэтому данный метод весьма прост. Кроме того, условия размещения легко изменить путем добавления, исключения или изменения определенных действий. Конечно, программа размещения не сможет обращаться с синтаксически неправильными «программами».

2. *Анализаторы программ*

Таким же образом можно подойти к решению следующих задач, для которых требуется различный анализ программ:

а) перечисление идентификаторов, используемых в каждом блоке;
б) учет числа реализаций различных типов предложений, операторов и т. п.;
в) оптимизация исходного кода. Здесь выходом будет другая версия программы на *том же самом* языке.

Вход в анализатор вовсе не обязательно должен быть написан на языке программирования, если его формат можно определить с помощью грамматики. Так, можно написать программу, которая читает календарь футбольных встреч вместе с результатами и обновляет его. Не нужно даже точно специфицировать вход; процедура ввода может, например, игнорировать пробелы и начало с новой строки.

Упражнения

6.1. Покажите, как нужно изменить пример из разд. 6.1, чтобы вместо четверок получать постфиксную нотацию. (В постфиксной нотации каждый знак операции появляется сразу вслед за своими операндами. Например, (инфиксное) выражение

$$(a+b)\times(c+d)$$

примет вид

$$ab+cd+\times$$

6.2. Приведите пример, показывающий, что включение в грамматику действия может воспрепятствовать ее преобразованию в LL(1)-форму.

6.3. Необходимо ли в примере из разд. 6.1. помещать в стек знак операции, если имеется только один знак операции с каждым приоритетом?

6.4. В некоторых языках приоритет знака операции может объявляться программистом. Как, по вашему мнению, синтаксический анализатор отреагирует на это?

6.5. Приведите примеры входов (помимо программ), которые синтаксический анализатор смог бы обработать.

6.6. Какие проблемы вы видите в написании полностью удовлетворительной программы «размещения»?

6.7. Объясните, как определить значения целочисленных констант во время лексического анализа с помощью включения в соответствующую грамматику действий.

6.8. Как вы считаете, какую форму примет таблица символов в языке Бейсик или Фортран?

6.9. Какие вы можете привести агрументы, чтобы разрешить описаниям появляться почти в любом месте в блоке?

6.10. В настоящей главе мы познакомили вас с таблицей разбора, которая может «вызвать» действия, выполняемые во время компиляции. Как в связи с этим следует видоизменить элементы таблицы разбора и драйвер LL(1), рассмотренный в разд. 4.5?

Глава 7

ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА

В этой главе мы подробно рассмотрим процесс проектирования компилятора. В гл. 6 приводились такие примеры, где лексический анализ не был необходим и всегда имелась информация, нужная во время компиляции (например, о типах). Это позволяло нам не затрагивать некоторые аспекты проблемы. Реальные языки программирования обычно намного сложнее, и именно определенные свойства компилируемого языка влияют в какой-то степени на общий проект компилятора. Проект также зависит и от того, какова среда, в которой будет работать компилятор, какой этап должен быть более эффективным — компиляция или прогон и какое внимание уделяется обеспечению хорошей диагностики или возможности исправления ошибок.

7.1. ЧИСЛО ПРОХОДОВ

Многие разработчики компиляторов находят идею однопроходного компилятора привлекательной отчасти потому, что не надо заботиться о связях между проходами, о промежуточных языках и т. д. Кроме того, нет трудностей в ассоциировании ошибок программы с исходным текстом, не существует проблем, связанных с тем, как один проход должен «залатать» ошибку, чтобы можно было передать для следующего прохода что-нибудь вразумительное. Однопроходные компиляторы обычно быстрее многопроходных, если только они не вовлечены чрезмерно в возвраты или предварительные просмотры символов в тех случаях, когда вопрос лучше решить, добавив еще один проход. Однако однопроходные компиляторы могут выдавать более медленный объектный код, так как маловерятно, что они смогут осуществлять обширную оптимизацию.

Некоторые языки, такие, как Фортран или Паскаль, полностью компилируются за один проход. Если же язык нельзя скомпилировать за один проход, однопроходные компиляторы пишутся для его подмножеств или диалектов. Обычное ограничение, которое вводится для этих компиляторов, заключается в том, что все идентификаторы и т. д. должны описываться (в тексте) до их использования. В некоторых ситуациях язык необходимо расширить, чтобы, например, при взаимно рекурсивных типах или процедурах объект можно было бы «описать» до того, как он будет полностью определен. Так, ряд компиляторов Алгола 60 требуют описания меток, а компилятор Алгола 68R, написанный в RSRE (Малверн, Англия), имеет «описание»

```
mode abc
```

указывающее, что выделенное слово **abc** есть вид, определяемый полностью позднее, а не специфицируемое пользователем обозначение опе-

рации (в Алголе 68 для этих целей также применяется выделенное слово). Средство предварительного описания в Паскале позволяет аналогичным образом обрабатывать взаимно рекурсивные процедуры за один проход.

Для диалектов языков, полная компиляция которых требует двух или более проходов, пишутся не только однопроходные компиляторы, но и многопроходные. Однако далее в этом разделе мы ограничим свою задачу, установив минимальное число проходов, необходимое для компиляции всех программ, написанных на определенном языке, и не будем вводить ограничения на язык или добавлять к нему средства суперязыка, помогающие разработчику компилятора.

Прежде всего следует несколько подробнее остановиться на том, что мы подразумеваем под проходом компилятора. Если какая-либо фаза (или несколько фаз) процесса компиляции требует прочтения исходного текста или трансляции его на некоторый промежуточный язык, то это обычно считается проходом. Для нахождения минимального числа проходов при компиляции мы не позволим за один проход выполнять произвольное число предварительных просмотров символов (хотя синтаксическому анализатору может понадобиться фиксированное число просмотров), так как в противном случае можно было бы считать, что почти любой язык компилируется за один проход. Не вся работа, выполненная компилятором, обязательно входит в один из проходов. Отдельные компиляторы могут реорганизовывать таблицы или сортировать виды в промежутках между проходами. Если какая-то фаза компиляции не сопровождается чтением исходного кода (или его версии), то ее не нужно считать проходом. Допустим, однако, что фаза компилятора должна прочитать скелет исходного текста, который дает просто структуру исходной программы, но без деталей. Принять ли это за проход или нет? Ответ, вероятно, должен быть положительным, поскольку, несмотря на отсутствие деталей, со структурной точки зрения прочитывается вся программа, и длина скелета окажется почти пропорциональной длине программы.

Проходы могут быть прямыми и обратными, т. е. за один проход исходный текст может читаться слева направо (как правило) или справа налево. Минимальное число проходов, необходимое для компиляции языка, зависит от того, все ли проходы прямые или часть из них — прямые, а часть — обратные. Идея обратного прохода кажется нам привлекательной с эстетической точки зрения, но в ряде операционных систем и в системах организации файлов осуществить ее нелегко.

Необходимость в более чем одном проходе обычно возникает в связи с тем, что какая-то требуемая компилятору информация отсутствует в соответствующий момент времени. Так, для компиляции

$$x+y$$

во многих языках нужно знать типы x и y, чтобы определить знак операции $+$, который может означать сложение либо двух целых чисел, либо двух вещественных чисел, либо одного вещественного и одного целого и т. д. Если определяющим реализациям x и y разрешено следовать за их прикладными реализациями, код обычно не может генерироваться за один проход. В Алголе 60 эта проблема не возникает, но при наличии взаимно рекурсивных процедур одна процедура неизбежно должна объявляться раньше другой. Допустим, например, что тело процедуры A содержит вызов или вызовы процедуры B, а процедура B

содержит вызов или вызовы процедуры *A*. Если процедура *A* объявляется первой, то компилятор не будет генерировать код для вызова *B* внутри *A*, не зная типа параметров *B* (если они есть), и в случае процедуры, возвращающей результат (функции), тип этого результата может потребоваться для идентификации, скажем, обозначения операции. Единственное разумное решение данной проблемы — позволить компилятору сделать дополнительный проход перед генерацией кода. Это даст возможность определить типы идентификаторов и т. д. и поместить их в таблицу символов с тем, чтобы ими могли воспользоваться последующие проходы. Теоретически Алгол 60 можно скомпилировать таким методом за два прохода, хотя многие более ранние компиляторы Алгола 60 [47] имели часто семь — восемь проходов. Причина, очевидно, заключалась в том, что максимальный объем памяти, требуемый компилятору, в любой момент времени должен был быть как можно меньшим. Существуют также и концептуальные причины для разделения различных аспектов процесса компиляции на отдельные проходы.

Многие языки нуждаются по меньшей мере в двух проходах по тем же причинам, что и Алгол 60. Компиляторы полного Алгола 68 обычно имеют четыре — восемь проходов. Интересно выяснить, сколько проходов действительно необходимо для компиляции этого языка. Как и в Алголе 60, для генерации кода здесь нужно знать вид (или тип) каждого идентификатора, константы и т. д., а, поскольку описания идентификаторов могут появляться после их употребления, определенно требуется не меньше двух проходов. Представим себе такой проход, который просматривает программу, отмечая виды всех идентификаторов и помещая эту информацию в таблицу символов, чтобы ею можно было воспользоваться при следующем проходе для генерации кода. Однако во время первого прохода нельзя распознать все описания, так как последовательность

abc *abc*

может быть описанием идентификатора *abc*, вида **ref abc** или выражения

monadic operator «**abc**» *identifier «abc»*

С целью устранения этой неоднозначности необходимо определить, является ли **abc** специфицируемым пользователем видом или обозначением операции. В любом случае мы будем иметь соответствующее описание **abc**, хотя и необязательно перед его прикладной реализацией в

abc *abc*

Указанная проблема разрешается с помощью еще одного прохода. В терминах Алгола 68 **abc** — это выделенное слово, а выделенные слова могут использоваться для представления специфицируемых пользователем видов или обозначений операций. Информация о выделенном слове (вид или обозначение операции) может быть дана в виде текста после того, как она потребуется проходу, помещающему сведения о видах в таблицу символов. Поэтому нужен еще один, предшествующий этому, проход, который строил бы таблицы, содержащие информацию о том, как используются выделенные слова в различных блоках программы.

Теперь оказывается, что для компиляции Алгола 68 требуется не менее трех проходов. Причина, по которой Алголу 68 может понадо-

биться более трех проходов, связана с перегрузкой символа открывающей скобки «(». Этот символ употребляется для обозначения начала условного предложения, вариантного предложения, процедуры, совместного или замкнутого предложения, вместо открывающей квадратной скобки при индексации массивов и т. д. Значение

(real *x*, *y*, int *a*, *b*) real

нельзя определить до тех пор, пока не будет прочитана закрывающая скобка. До этого момента фрагмент мог быть либо началом процедуры, либо замкнутого предложения (или условного или вариантного предложения). Как мы видели в разд. 5.5, грамматики для замкнутого и условного предложений можно соединить, что нетрудно сделать также для замкнутого предложения и процедуры (или даже для всех четырех упомянутых вариантов). В результате анализатор не будет знать, имеет ли он дело с замкнутым предложением или процедурой, пока различие не станет очевидным (в вышеприведенном примере при встрече закрывающей скобки). Поэтому с точки зрения разбора перегрузка символа открывающей скобки не вызовет затруднений. Однако если в анализатор должны вкладываться действия для генерации кода или обновления таблиц, возникают трудности. Предполагается, что должны выполняться разные действия после считывания

real *x*

в зависимости от того, часть ли это замкнутого предложения или процедуры. Если данный фрагмент встречается в замкнутом предложении, то он служит описанием идентификатора, и в таблицу символов вносятся соответствующие элементы. Если же он встречается в процедуре, то берется на заметку вид одного из параметров, который в свою очередь влияет на вид самой процедуры.

Требует ли открывающая скобка в процедуре дополнительного прохода для компиляции? Этого можно избежать, объединив все вышеописанные действия в одно, которое выполнялось бы в любом случае, когда возникает сомнение относительно того, является ли фрагмент описанием идентификатора или формальным параметром. Однако такой метод потребовал бы исключения из таблицы этих элементов при устранении неоднозначности либо лексический анализатор должен был бы иметь возможность действовать с предварением, чтобы распознавать открывающие скобки процедур, которые он мог бы заменить каким-либо другим символом. Несмотря на то что процедура может иметь произвольно большое число параметров, число предварительно просматриваемых символов не должно быть произвольно большим. На практике оно, вероятно, довольно мало. Проблема, по существу, заключается в передаче информации обратно по исходному тексту, и, помимо использования прохода с предварительным просмотром или возвращением, можно применить тот же прием, что и при разрешении проблемы прикладных реализаций идентификаторов, встречавшихся ранее соответствующих определяющих реализаций. Иными словами, на одном из первых проходов нужно составить таблицу, содержащую информацию для использования в последующих проходах. В этой таблице можно выделить место для каждой открывающей скобки в программе, и после определения ее типа таблицу обновлять.

Приведем пример типичного фрагмента программы, в котором появляются открывающие скобки:

(square := **(int** q**)** **int** : $q \uparrow 2$;
$a := (b|c|d) \cdots$

обозначающие соответственно **begin**, начало процедуры, и **if**.

Алгоритм заполнения таблицы скобок следующий. Всякий раз, когда встречается открывающая скобка, выделяется новый элемент таблицы, и его адрес помещается в стек. Если это условное или вариантное предложение, тип открывающей скобки распознается при встрече второй черты | (при ее наличии) или закрывающей скобки (вариантное предложение имеет не менее двух элементов). Элемент таблицы, соответствующий адресу, помещенному в верхней части стека, обновляется, а при встрече закрывающей скобки этот адрес будет удален. Если к моменту встречи закрывающей скобки выясняется, что данная конструкция не представляет собой ни условное, ни вариантное предложение, то решение откладывается до встречи с двоеточием (обозначающим процедуру), и тогда обновляется соответствующий элемент таблицы, а стек уменьшается на один элемент. Эту фазу можно объединить с другими, и в тех случаях, когда значения выделенных слов известны во время ее выполнения, замкнутые предложения и процедуры могут уже быть определены к моменту прочтения закрывающей скобки, так как мало вероятно, чтобы последний элемент замкнутого предложения был описанием.

С точки зрения генерации кода способность различать условные и замкнутые предложения при чтении открывающей скобки не столь важна. Но для диагностики (и, очевидно, для сохранения минимального размера таблицы разбора) это свойство весьма полезно. Рассмотрим пример:

$$(a := b;\ L : c := d;\ a = c|x|y)$$

В этом фрагменте есть синтаксическая ошибка, поскольку в условном предложении между **if** (представленным открывающей скобкой) и **then** (представленным посредством черты |) метки не допускаются. Однако если открывающая скобка не распознается как **if**, сообщение об ошибке не выдается до тех пор, пока не встретится «|».

Организацию таблицы скобок можно заменить дополнительным обратным проходом непосредственно перед синтаксическим анализом. Этот проход воспользуется стеком, как уже описывалось ранее, только вместо внесения в таблицу информации он будет фактически исправлять исходный текст, заменяя открывающие скобки другими символами, показывающими их значение. Допускается также один прямой проход, если исходный текст хранится в каком-либо виде в памяти с прямым доступом, а стек служит для запоминания адресов открывающих скобок до тех пор, пока их значение не проясняется, и на этой стадии исходный текст обновляется включением полученной информации.

Проход, выясняющий значение скобок, применяют также для проверки исходного текста. В Алголе 68 **if** должно иметь соответствующее ему **fi**, а открывающая скобка — закрывающую, т. е. последовательности

if .....)

или

(.....**fi**

являются недопустимыми (то же относится к вариантному предложению и т. д.). Скобочный проход должен проверять такое соответствие и просто заменять открывающие и закрывающие скобки на **if, fi, case** и т. д., что поможет сократить размер грамматики (а следовательно, и таблицы разбора).

В Алголе 68 квадратные скобки обычно употребляются в связи с массивами как для определяющих, так и для прикладных реализаций, например

$$[1 : n]\,\mathbf{real}\ x$$

или

$$x[4]$$

Однако в полной реализации языка квадратные скобки разрешается заменять (согласующимися) круглыми скобками. Это неизбежно приводит к осложнениям в том, что касается стадии анализа в компиляции. Рассмотрим сначала прикладные реализации. Фрагмент

$$S(i,j)$$

в зависимости от вида S может быть либо «вырезкой» массива, либо вызовом (процедуры). Конечно, вид S может быть неизвестен, когда этот фрагмент текста встречается впервые, поэтому он остается неоднозначным до тех пор, пока не поступит полная информация о видах. (Вообще S можно заменить предложением, представляющим строку (массив) или процедуру.) В любом случае эта информация потребуется для последнего прохода в стадии анализа. Поэтому он сможет определить значение $S(i,j)$ и генерировать код, если это потребуется. Приведенный пример показывает, в каком смысле разбор Алгола 68 зависит от вида, т. е. конструкции не могут быть полностью распознаны без информации о видах.

Обратимся теперь к определяющим реализациям, например к

$$(m : n)\ \mathbf{int}\ S$$

К тому времени, как будет прочитано **int** (или какой-либо другой описатель, стандартный либо специфицируемый пользователем), анализатору становится ясно, что это касается описания массива. Однако с точки зрения выполнения соответствующих действий было бы полезно распознать открывающую скобку, по существу, как «квадратную». Это действие можно встроить в скобочный проход, рассмотренный ранее.

Стадия анализа в компиляции часто состоит из нескольких проходов исходного текста (или его трансляции), и каждый проход выполняет одну или две фазы процесса компиляции. Например, в типичном компиляторе Алгола 68 фаза лексического анализа и фаза идентификации выделенных слов могут выполняться параллельно за один и тот же проход. В правильно построенном компиляторе, однако, эти две фазы в представлении разработчика компилятора разделены, даже несмотря на то, что во время компиляции они выполняются параллельно. Один из способов добиться этого — написать лексический анализатор в виде процедуры с вызовом фазы выделенных слов всякий раз, когда должен считываться новый символ. В этом случае фаза выделенных слов увидит исходный текст так, как если бы лексический анализ проводился в предыдущем проходе, а фазе лексического анализа ничего не будет известно о механике работы фазы выделенных слов. Такой

метод почти совершенен в отношении модульности, документирования и ведения, а также надежности.

Каждый проход в стадии анализа может направляться синтаксисом, т. е. основываться на грамматике. Однако для разных проходов грамматики могут быть разными. Например, если проход имел дело только с описаниями для выяснения типов, приоритет знаков операций здесь не существен, и разница между вызовами и вырезками не имеет значения. Тем не менее от последнего прохода в стадии анализа требуется образование полного синтаксического дерева для генерации кода, и нужно, чтобы он базировался на полной контекстно-свободной грамматике. Грамматика, использовавшаяся в более раннем проходе, соответствует сверхмножеству языка, генерированного грамматикой, которая применялась в более позднем проходе. Грамматика раннего прохода, имеющая меньше ограничений, обычно меньше по объему (как и соответствующая ей таблица разбора), чем грамматика более позднего прохода.

Контекстно-зависимые требования языка, как правило, представляются действиями, включенными в контекстно-свободную грамматику. Ими могут быть внесение информации в таблицу символов или таблицу выделенных слов и поиск вида либо другой информации в одной из этих таблиц. Такие действия соответствуют атрибутам в атрибутивной грамматике или метапонятиям в двухуровневой грамматике. Действие, соответствующее каждому атрибуту или метапонятию в грамматике, появляется в одном из проходов на стадии анализа. Например, проход, который распознает описания идентификаторов, включает действия по внесению в таблицу символов информации о видах, а последующий проход включает действия поиска этой информации в таблице символов. Проверку контекстно-зависимых аспектов языков можно представить распределенной по ряду проходов в компиляции аналогично тому, как это имеет место с контекстно-свободными аспектами. Существует алгоритм, позволяющий определить минимальное число проходов, необходимых для анализа заданной атрибутивной грамматики [9].

Можно назвать еще ряд причин, по которым компиляторы Алгола 68 (или другие) могут иметь дополнительные проходы. Это связано с используемой для генерации машинного кода моделью, с тем, какие требуются оптимизации, и должны ли мы сделать компилятор переносимым. Некоторые из указанных вопросов будут обсуждаться в гл. 10 и 11.

Мы определили, что компилятору Алгола 68 понадобится не менее трех проходов, чтобы проанализировать все программы на этом языке. Но не каждой программе для анализа нужны все три прохода. Например, в программе могут отсутствовать прикладные реализации идентификаторов, встречающиеся перед их определяющими реализациями, либо специфицируемые пользователем виды или обозначения операций. В таком случае программа *могла бы* быть скомпилирована за один проход. Обработка ее тремя проходами, в то время как можно обойтись только одним, представляется излишней. Однако компилятору не известно в начале компиляции (если ему не сообщит пользователь), требуются ли проходы для построения таблиц символов и т. п. или ему достаточно одного прохода. Он узнает об этом лишь по завершении анализа. В Манчестерском компиляторе Алгола 68 для MU5[5] принято следующее решение. Считается, что анализ может быть выполнен за один проход, если это не окажется невозможным.

Рассмотрим следующий пример:

```
begin int p;
 begin proc q = void:
  begin ··· p; ···
   end;
   proc p = void:
```

Компилятор может пытаться завершить анализ приведенной программы за один проход. Информация заносится в таблицы по мере поступления, и компиляция выполняется при допущении, что все было описано заранее. Этот процесс продолжается до тех пор, пока это условие не нарушится. В нашем примере первоначально считается, что прикладная реализация p в теле процедуры q относится к p, описанному во внешнем блоке. Когда же описание p встретится в *том же самом* блоке, станет очевидным, что было сделано ложное допущение, и от попытки завершить анализ за один проход придется отказаться. Однако компилятор продолжит чтение исходного текста, чтобы собрать как можно больше информации для таблицы символов. Таблица символов потребуется при следующем проходе, когда снова будет сделана попытка проанализировать всю программу полностью, на этот раз с помощью некоторой табличной информации. Ясно, что из-за неправильного допущения, принятого при первом проходе, могут появиться ложные сообщения об ошибках, но они не передаются сразу пользователю.

Преимущество рассмотренной схемы заключается в том, что программы анализируются за минимально возможное число проходов (часто за один); максимальное же число проходов предоставляется только тем программам, которые в этом нуждаются.

В компиляторе Алгола 68, написанном в Стратклайдском университете, при первом проходе выполняется лишь лексический анализ. Следовательно, к концу первого прохода известно, появляются ли в программе символы **op** или **mode**. Если не появляется **op**, то выделенные слова в программе (кроме таких слов языка, как **begin, for** и т. д.) должны соответствовать специфицируемым пользователем видам, а если не появляется **mode** — обозначениям операций. При наличии этой информации можно часто пропускать проход, который идентифицирует слова.

Мы ограничили наше обсуждение преимущественно Алголом 68. Однако подобный анализ, можно провести и для определения минимального числа проходов при компиляции других языков. Как выяснилось, во многих компиляторах предусмотрено большее число проходов, чем это требуется теоретически для компиляции программ на исходном языке.

7.2. ПРОМЕЖУТОЧНЫЕ ЯЗЫКИ

За написание компилятора, содержащего более одного прохода, мы вынуждены расплачиваться тем, что нам приходится проектировать промежуточные языки, на которых должен представляться исходный текст между проходами (т. е. в виде выхода из одного прохода и входа в следующий). Эту задачу можно решать на двух уровнях: на уровне исходного текста и на уровне лексического анализа. Как мы видели в гл. 3, одно из основных действий, выполняемых во время лексического анализа, — замена объектов переменной длины, например идентификаторов, констант и т. д., символами фиксированной длины. Наи-

простейшей формой этих символов являются целые числа. Поэтому исходный текст после лексического анализа может состоять из последовательности целых чисел, одни из которых соответствуют словам языка, а другие представляют указатели на (содержимое) таблицы идентификаторов или констант и т. д. Такая промежуточная программа сама по себе не содержит всей информации, имеющейся в оригинальном исходном тексте, но вместе с таблицами, организованными во время лексического анализа, она, по существу, окажется эквивалентной исходному тексту.

Напротив, промежуточный язык, выдаваемый при одном из последних проходов, будет гораздо ближе к машинному коду. Примером тому могут служит четверки. Такой промежуточный язык рассматривается в следующем разделе.

Что же сказать о коде, который выдается вторым и третьим проходами типичного пятипроходного компилятора? На какой промежуточный язык он больше похож — первого типа или второго? Вероятно, на язык первого типа. Действительно, промежуточные языки, выдаваемые первыми проходами, могут быть одинаковыми. Ведь первые проходы служат в основном для построения таблиц, которыми пользуются последующие проходы, и несколько проходов могут фактически читать одну и ту же версию исходного текста, хотя они и выбирают для таблиц разную информацию. Возможен и такой вариант, что промежуточный *язык* останется тем же, но последующие проходы будут обогащать исходный текст, включая в него дополнительную информацию (например, значения скобок и т. п.), которая потребуется синтаксическому анализатору.

Можно считать, что многопроходный компилятор выполняет ряд предварительных проходов, строящих таблицы символов и т. д. и (возможно) обогащающих исходный текст для того, чтобы анализ (а часто и генерация кода) мог быть завершен за один проход.

7.3. ОБЪЕКТНЫЕ ЯЗЫКИ

Иногда можно разделять не зависящие и зависящие от машины части компилятора, что весьма желательно в тех случаях, когда его предполагают сделать переносимым. С этой целью нужно определить так называемый объектный язык, который является посредником между двумя частями: не зависящей от машины частью компилятора, транслирующей исходный код в объектный язык, и зависящей от машины частью, транслирующей объектный язык в конкретный машинный код. Если бы мы могли использовать *один и тот же* объектный язык (и общий язык реализации), задача реализации m языков на n машинах свелась бы к написанию $m+n$ частей программного обеспечения, т. е. m не зависящих от машины «фронтальных частей» компилятора и n зависящих от нее «хвостов» компилятора, а не $m \times n$ полных компиляторов. Объектный язык можно также представить себе в виде машины, обычно называемой абстрактной, так как фронтальная часть компилятора видит его как объектную машину. Существует целый ряд таких универсальных объектных языков или абстрактных машин, например Янус (JANUS) [48], но все же эта идея, по-видимому, не наш-

ла достойного воплощения[1]. Чтобы охватывать широкий класс языков, объектный язык должен быть написан на довольно низком уровне с целью его эффективной реализации на всех машинах. Однако было бы лучше, если бы объектный язык предназначался для трансляции *конкретного* языка высокого уровня или для реализации на *конкретной* машине. Так, большинство компиляторов, работающих на машине MU5, в качестве объектного языка применяют CTL (общий объектный язык). С другой стороны, на некоторых машинах, где в качестве объектного языка употреблялись OCODE или INTCODE, реализован BCPL [49], а Бранкар и др.–[10] спроектировали объектный язык для Алгола 68, который можно реализовать на многих машинах. В последнем случае фронтальной части компилятора нужно иметь кое-какую информацию об объектной машине, в частности сведения о том, сколько места требуется для целочисленного значения, вещественного значения и т. д. Это связано с тем, что распределение памяти происходит до выработки объектного языка. Можно, конечно, распределять память и без этой зависимой от машины информации, но тогда обращение с адресами во время прогона неоправданно усложнится.

В настоящей главе мы рассмотрели некоторые аспекты проектирования компиляторов: разбиение их на фазы и проходы и проектирование промежуточных языков. Следующие главы будут посвящены проектированию таблиц символов и видов, и моделям распределения памяти.

Упражнения

7.1. Укажите причины, по которым языки не всегда проектируются так, чтобы их можно было скомпилировать за один проход.

7.2. Обоснуйте, почему может оказаться желательным включить в компилятор большее число проходов, чем это строго необходимо для компиляции заданного языка.

7.3. Прокомментируйте утверждение о том, что однопроходный компилятор может быть более удобным с диагностической точки зрения.

7.4. Назовите языковое средство, удобное для применения при проходе компилятора, выполняемого в обратном порядке.

7.5. Приведите аргументы «за» и «против» удобочитаемых промежуточных языков.

7.6. Как вы думаете, должен ли компилятор выдавать сообщение о проходе, в котором он обнаружил ошибку программирования?

7.7. Компилятор Алгола 68 для MU5 имеет «переменное» число проходов. Приведите аргументы против такого подхода.

7.8. Как может повлиять на проект компилятора машина, для которой он предназначается?

7.9. Если бы основной целью при проектировании компилятора было обеспечение его переносимости, то как это могло бы отразиться на его общей структуре?

7.10. Если бы основной целью при проектировании компилятора было обеспечение его надежности, то как это могло бы отразиться на его общей структуре?

---

[1] Можно указать на один из ранних объектных языков АЛМО, разработанный и использованный для создания переносимых компиляторов в Институте прикладной математики АН СССР в 1969 — 1973 гг. с языков АЛГАМС, Алгол 60, Фортран на ЭВМ М-20, БЭСМ-6, и ряд специализированных ЭВМ. — *Примеч. ред.*

Глава 8

ТАБЛИЦЫ СИМВОЛОВ И ВИДОВ

Информацию о типе (или виде) идентификаторов и т. п. синтаксический анализатор хранит с помощью таблиц символов. Ими также пользуется генератор кода для хранения адресов значений во время прогона. В языках, имеющих конечное число типов, информацией о типе может быть просто целое число, представляющее этот тип, а в языках, имеющих потенциально бесконечное число типов (видов), таких, как Алгол 68 или Паскаль, — указатель на таблицу видов, элементами которой являются структуры, представляющие вид. В настоящей главе мы сначала рассмотрим таблицы символов, а затем таблицы видов.

8.1. ТАБЛИЦЫ СИМВОЛОВ

Таблица символов для таких языков, как Фортран или Бейсик, довольно проста. В Фортране, например, имеется только конечное (и небольшое) число типов, и каждый из них может быть представлен целым числом. Например, тип «целый» представляется посредством *1*, а тип «вещественный» — посредством *2*. В этом случае таблица символов имеет вид массива с элементами

identifier, type

В Фортране идентификаторы не должны превышать шести (а в Бейсике двух) литер, так что в таблице символов хранятся сами идентификаторы, а не какое-либо их представление, полученное после лексического анализа.

С таблицей символов ассоциируются следующие действия:

1. Идентификатор, встречающийся впервые (т. е. его еще нет в таблице), помещается в таблицу вместе со своим типом. Если идентификатору не предшествует явное описание, такое, как

REAL X

его тип, можно вывести на основании начальной буквы (с *I* по *N* включительно подразумевается целый, в других случаях — вещественный).

2. Если встречается идентификатор, который уже был помещен в таблицу, его тип определяется по соответствующей записи в этой таблице.

В соответствии с такой моделью идентификаторы будут появляться в таблице в том же порядке, в каком они встречаются впервые в программе. Всякий раз, когда анализатору встречается идентификатор, он проверяет, есть ли этот идентификатор уже в таблице, и при его отсутствии в конец таблицы вносится соответствующая запись. Если иден-

тификатора в таблице *нет*, то требуется ее полный просмотр (элемент за элементом); но даже в тех случаях, когда идентификатор находится в таблице, поиск в среднем охватывает ее половину. Такой поиск в таблице обычно называют *линейным*. Линейный поиск в больших таблицах может оказаться длительным процессом. Если бы идентификаторы были расположены в алфавитном порядке, то поиск выполнялся бы гораздо быстрее за счет последовательного разбиения таблицы пополам до тех пор, пока соответствующий идентификатор не будет найден или не будет показано его отсутствие (двоичный метод поиска). Однако на сортировку записей по порядку каждый раз при добавлении новой записи уходило бы очень много времени, что, по-видимому, существенно снизило бы достоинства этого метода (конечно, если число поисков не превышало бы намного число включений в таблицу). Другое решение проблемы — использование метода *хеширования*.

Чтобы познакомиться с идеей хеширования, предположим, что Фортран допускает использование идентификаторов, содержащих только одну букву. Тогда число возможных идентификаторов составит 26, и таблица символов примет вид массива, состоящего из 26 элементов — по одному на каждый идентификатор. Это означает, что для проверки наличия записи в таблице для какого-либо конкретного идентификатора никакого поиска не требуется, поскольку любая запись для идентификатора *A* служит первым элементом таблицы и т. д. Такой метод подошел бы для Бейсика, в котором все идентификаторы имеют обычно одну из следующих форм:

letter
letter digit
letter dollar

или

letter percent

Это дает только 338 возможных идентификаторов. Для Фортрана число возможных идентификаторов, хотя и конечно, но очень велико, а для Алгола 60, Алгола 68 или Паскаля — бесконечно.

В общем случае используется массив, состоящий из бóльшего числа элементов, чем максимальное число ожидаемых идентификаторов в программе, и определяется отображение (называемое *функцией хеширования*) каждого *возможного* идентификатора на элемент массива. Это отображение не является, конечно, отображением один к одному, как у нас было ранее, а множество (или бесконечное число) идентификаторов отображается на один и тот же элемент массива. Всякий раз при появлении идентификатора в Фортране проверяется наличие записи в соответствующем элементе массива. При отсутствии записи этот идентификатор еще не находится в таблице символов, и можно сделать необходимую запись. Если этот элемент не пустой, проверяется, соответствует ли запись в нем данному идентификатору, и когда выясняется, что соответствия нет, точно так же исследуется следующий элемент и т. д. до тех пор, пока не обнаружится пустой элемент или запись, соответствующая данному идентификатору. Таким образом, таблица символов просматривается (линейно), начиная с записи, полученной с помощью функции отображения, до тех пор, пока не встретится сам идентификатор или пустой элемент, указывающий на то, что для этого идентификатора записи не существует. Такой массив обычно называет-

ся *таблицей хеширования*, а функция отображения — *функцией хеширования*. Таблица хеширования обрабатывается циклично, т. е. если поиск не завершается к моменту достижения конца таблицы, он должен быть продолжен с ее начала.

Простая функция хеширования подразумевает использование первой буквы каждого идентификатора для его отображения на элемент 26-элементного массива. Идентификаторы, начинающиеся с *A*, отображаются на первый элемент, начинающиеся с *B* — на второй и т. д. После встречи с идентификаторами

CAR DOG CAB ASS EGG

Таблица 8.1

1	*ASS*
2	
3	*CAR*
4	*DOG*
5	*CAB*
6	*EGG*
7	
8	
9	
.	
.	
.	

таблица принимает вид табл. 8.1; позиции идентификаторов зависят от порядка их внесения в таблицу. Если идентификатор не может быть внесен в ту позицию, которая задается функцией хеширования, происходит так называемый *конфликт*. Чем больше таблица (и меньше число идентификаторов, отображающихся на каждую запись), тем меньше вероятность конфликтов. При наличии в программе только вышеперечисленных идентификаторов и использовании рассмотренной функции хеширования таблица заполнялась бы неравномерно, и имела бы место *кластеризация*. Программисты иногда употребляют идентификаторы, начинающиеся с определенных букв алфавита. В этом случае функция хеширования может создавать конфликты, и поэтому она весьма проста. Функция, которая зависела бы от последней литеры идентификатора, вероятно, меньше способствовала бы кластеризации или же функция хеширования могла бы использовать все литеры в идентификаторе для вычисления соответствующего элемента таблицы. Конечно, чем сложнее функция, тем больше времени потребуется для ее вычисления при внесении в таблицу идентификатора или при поиске конкретного идентификатора в таблице, и это следует соотносить со временем, которое экономится за счет более коротких поисков в результате меньшей кластеризации.

До сих пор мы выходили из конфликтов, просто проверяя один за другим элементы таблицы (цикличным образом) до тех пор, пока не находили идентификатор или пустой элемент. Чтобы найти другой элемент в таблице, когда тот, на который указывает функция хеширования, уже заполнен, можно воспользоваться и иными методами. Нам нужно правило, последовательное применение которого позволило бы (при необходимости) просмотреть все записи таблицы, прежде чем какая-либо из них встретится вторично. Действие, выполняемое функ-

цией хеширования, обычно называется *первичным хешированием*, а вычисление последующих адресов в таблице — *вторичным хешированием*, или *перехешированием*. Функция перехеширования, рассматриваемая до сих пор, предполагает просто добавление единицы (циклично) к адресу таблицы. Эта функция, как и все функции перехеширования, обладает следующим свойством: если n — некий адрес в таблице, а p — число элементов в таблице,

$$n,\ rehash\ (n),\ rehash^2\ (n) \ldots rehash^{p-1}\ (n)$$

все являются разными адресами, и

$$rehash^{p}(n) = n$$

Описанная выше функция перехеширования определяется процедурой

```
proc rehash = (int n) int:
if n<p then n+1 else 1 fi
```

У этой функции есть тенденция создавать в таблице кластеры в той же мере, в какой вероятность кластеризации возрастает в связи с перехешированием. Весьма желательно, чтобы функция перехеширования могла бы находить адрес подальше от того, с которого она начала. Если бы число элементов в таблице было простым числом, то вместо единицы функция перехеширования добавляла бы к адресу любое положительное целое число h, такое, что $h<p$; в результате мы бы имели

```
proc rehash1 = (int n) int:
if n+h≤p then n+h
else n+h−p fi
```

Подходящее значение h сводило бы кластеризацию к минимуму. Так как p — простое число, функция перехеширования выдаст последовательно все адреса в таблице, прежде чем она повторится. Чтобы это положение сохранялось, p не обязательно даже должно быть простым числом, но h и p должны быть относительно простыми, т. е. не иметь других множителей, кроме единицы.

Более подробную информацию о хешировании и перехешировании читатель может найти в [24]. Мы объяснили лишь основной принцип, а теперь рассмотрим другие подходы к решению данной проблемы. Например, можно полностью избежать перехеширования путем сцепления элементов, которые переполняют таблицу (см. рис. 8.1), однако при этом придется выделять место для указателей.

Еще одна структура для записи упорядоченных элементов — *бинарное дерево*. Бинарное дерево, показанное на рис. 8.2, состоит из пяти элементов (или вершин), каждый из которых содержит идентификатор, его тип и т. д., и двух указателей на другие вершины. Это дерево может и не иметь указателей (в случае вершин, содержащих *BUS*, *HADDOCK* и *MOUSE*).

По определению бинарное дерево включает конечное множество вершин. Оно может быть пустым или состоять из корня (вершина, содержащая *LEMON*) и двух отдельных бинарных деревьев, которые называются соответственно левым и правым поддеревьями корня. Бинарное дерево, не имеющее вершин, называется пустым бинарным деревом и на схеме обозначается как перечеркнутый прямоугольник. Поддерево вершины обозначается с помощью указателя.

О бинарном дереве, изображенном на рис. 8.2, говорят, что оно имеет глубину 2, а вершина, содержащая *BUS*, находится на расстоя-

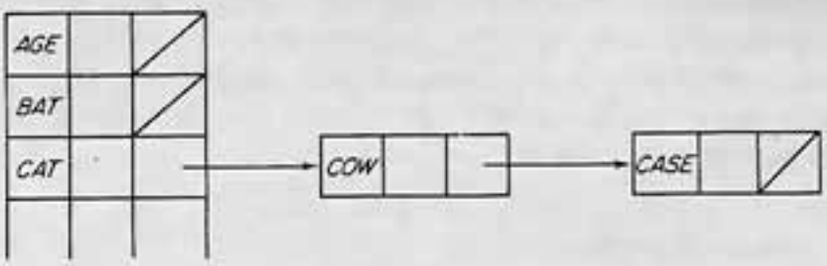


Рис. 8.1

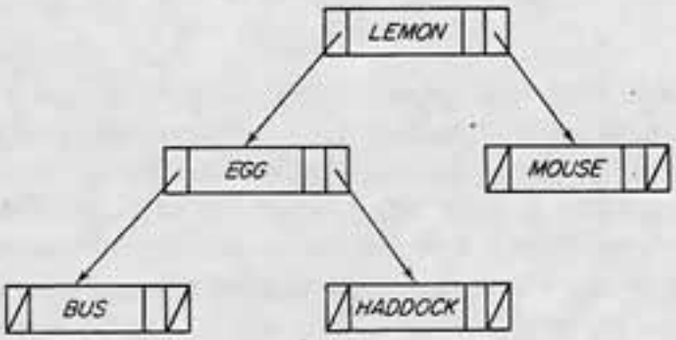


Рис. 8.2

нии 2 от вершины, содержащей *LEMON*. Вершину, содержащую *EGG*, называют наследником вершины, содержащей *LEMON*.

Это бинарное дерево упорядочено (в алфавитном порядке) слева направо, точнее, его вершины располагаются по алфавиту в порядке их обхода изнутри. Кнут [37] приводит следующий (рекурсивный) алгоритм для обхода дерева изнутри: пересечь левое поддерево, пройти корень, пересечь правое поддерево. К бинарному дереву всегда можно добавить новую вершину, поместив ее в соответствующее место. Время поиска вершины зависит только от глубины дерева. Это время минимально (в среднем) для дерева, содержащего заданное число вершин, если оно уравновешено, т. е. если расстояния от всех конечных вершин (вершин, не имеющих наследников) до корня отличаются не более чем на одну вершину. Дерево на рис. 8.2 уравновешено, а дерево на рис. 8.3 — нет.

Бинарное дерево можно строить, добавляя по одной вершине. Первой вершиной служит корень. К нему справа или слева добавляется следующая вершина и т. д. Уравновешенность полученного дерева зависит от порядка включения вершин. Если дерево не уравновешено, то должно существовать другое уравновешенное бинарное дерево с тем же порядком обхода вершин. Существуют довольно эффективные алгоритмы уравновешивания деревьев (замены неуравновешенного дерева уравновешенным), и в некоторых случаях имеет смысл производить уравновешивание после каждого добавления вершины. В основном это определяется относительной частотой поисков и добавлений.

Расплачиваться за использование бинарного дерева в качестве таблицы символов или словаря нам приходится дополнительным объе-

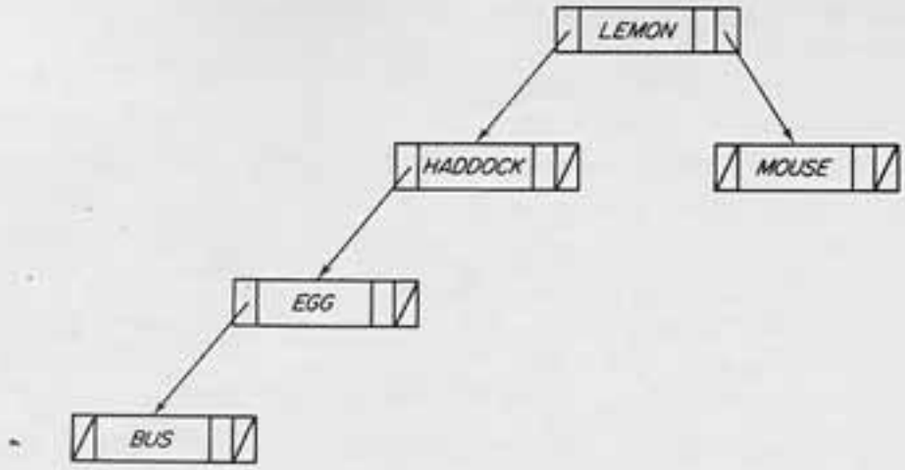


Рис. 8.3

мом памяти, требуемым для указателей. Тем не менее при сравнении бинарных деревьев и таблиц хеширования как таблиц символов следует помнить, что таблица хеширования может быть эффективной и не слишком кластеризованной, если она рассчитана на значительно большее (возможно на 50%) число записей, чем то, которое она, по предположению, будет содержать.

Часто (в зависимости от компилируемого языка) проблема построения и поиска в таблицах символов в компиляторе оказывается гораздо сложнее, чем мы ее себе представляем. Как отмечалось в гл. 6, большинство современных языков имеет блочную структуру, причем различные блоки могут содержать реализации одного и того же идентификатора, но с разными значениями. Например, мы можем написать следующую программу на Алголе 68:

```
begin int m := 4;
   begin char m = "A";
   print (m)
end;
begin int m := 1;
   print (m)
end;
print (m)
end
```

и выходом будет

A 1 4

в соответствии с правилами соотнесения прикладных и определяющих реализаций идентификаторов в языке. Что касается таблицы символов, то она должна содержать записи, соответствующие каждому из трех объявлений *m* в программе, и быть построена таким образом, чтобы необходимую запись для *m* можно было найти всякий раз, когда производится поиск при встрече с прикладной реализацией *m*.

Осложнение возникает и в связи с тем, что во многих языках, таких, как Алгол 68 (но не Паскаль), идентификаторы могут появляться в тексте, прежде чем они описываются. В результате компилятор строит таблицу символов за один проход, и только при последующем проходе или проходах в ней осуществляется поиск. Допустим, что *A* помещает

в таблицу символов информацию для ее использования последующим проходом *B*. Проход *A* имеет таблицу блоков, которая может быть массивом структур типа

mode btable = struct (**int** *levno*, **ref list** *idlist*)

где **list** — это

mode list = struct (**int** *tag*, *type*, **ref list** *next*)

tag — целочисленное представление идентификатора (в предположении, что лексический анализ уже прошел), а *type* — его тип (или вид). Каждый блок соответствует элементу массива *btab*, описанному как

[*1* : *n*]**btable** *btab*

Поле целого числа *levno* обозначает уровень каждого блока, т. е. число внешних блоков, включающих данный блок. Каждый блок имеет связанный с ним список идентификаторов (возможно, и других определяемых пользователем объектов) и их типов. Если число типов в языке конечно, для представления типа может использоваться целое число. В противном случае может потребоваться структура или указатель на таблицу видов (см. следующий раздел).

Во время компиляции при входе в блок при проходе *A* выполняется инициализация элемента *btab* следующим образом:

btab [*bn* **plusab** *1*] := (*ln* **plusab** *1*, **nil**)

где *bn* — абсолютный номер блока, а *ln* — номер уровня. При выходе из блока выполняется следующее действие:

ln **minusab** *1*

Когда встречается описание идентификатора *id* типа *typeid*, выполняется действие

idlist **of** *btab* [*bn*] := **heap list** := (*id*, *typeid*, *idlist* **of** *btab* [*bn*])

где **heap list** — «генератор», выделяющий место для другого элемента **list**. Порядок идентификаторов в **list** обратен тому порядку, в котором соответствующие описания появлялись в программе. Как указывалось в разд. 6.2, можно проверить, не описывался ли уже идентификатор в блоке.

Для приводимой ниже программы:

```
begin int a, b, c;
      .....
блок 0 { блок 1 { begin real d, e, f;
                  .....
                  end;
         блок 2 { begin int a, b; real c, d;
                  .....
                  end
         end
```

соответствующая таблица будет выглядеть так, как показано на рис. 8.4. В Алголе 68 можно также описывать объекты, их виды и знаки операций, специфицируемые пользователем, и в каждой записи *btab* должны также содержаться списки этих объектов и их видов.

В конце прохода *A* информация из таблицы символов, собранная со всех блоков, выводится в файл, причем информация из каждого

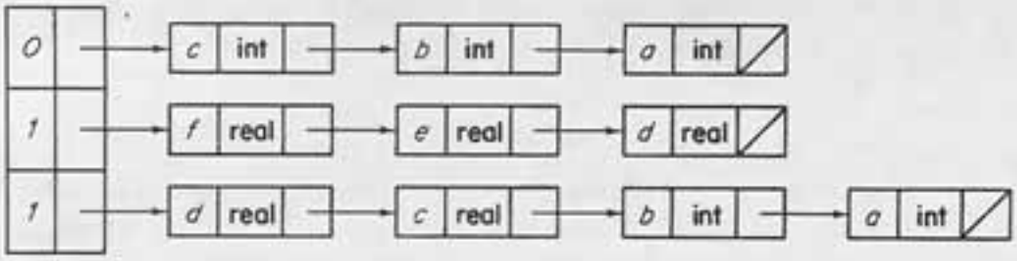


Рис. 8.4

блока выводится в том порядке, в каком осуществлялся первоначальный вход в блоки, т. е. в порядке появления блоков в таблице блоков.

В проходе *B* при входе в каждый блок выполняется считывание информации из таблицы символов, соответствующей данному блоку, и привязка ее к информации, относящейся к включающим блокам. Например, в вышеприведенной программе при входе в блок 2 таблица символов примет вид показанной на рис. 8.5, где другие объекты, специфицируемые пользователем (помимо идентификаторов), в таблицу не включены. Вместо списка идентификаторов, ассоциируемых с каждым блоком, используется массив, так как число описанных в каждом блоке идентификаторов известно из предыдущего прохода.

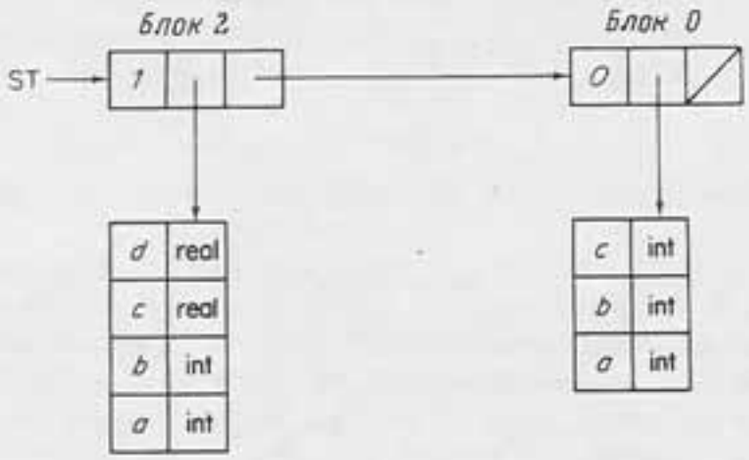


Рис. 8.5

При выходе из блока соответствующая ему информация может быть исключена из таблицы символов, если она не требуется при последующем проходе. Описание вида для таблицы символов при проходе *B* может быть следующим:

mode st = struct (**int** *levno*, **ref** [ ]**stentry** *ste*, **ref st** *next*)

где **stentry** описывается как

mode stentry = struct (**int** *tag*, *type*)

Идентификатор *ST* описывается (и инициализируется) так:

ref st *ST* := **nil**

При входе в блок выполняется следующее действие:

ST := (*ln* **plusab** *1*, *readids*, *ST*)

где *ln* — номер уровня блока, а *readids* — процедура чтения идент-

фикаторов и их типов и представления соответствующего массива структур, а при выходе из блока — действие:

```
ST := next of ST.
In minusab /
```

Когда в проходе *B* встречается прикладная реализация идентификатора, поиск в таблице символов осуществляется с начала списка и до конца. Таким образом, запись, соответствующая данному описанию идентификатора, всегда обнаруживается первой. (Можно допустить, что, если идентификатор описывается дважды в одном и том же блоке, это выявляется в проходе *A*). Такой метод подразумевает линейный поиск по (возможно, всем) записям в таблице символов. Можно также воспользоваться таблицей хеширования, поместив в нее все идентификаторы, описанные в каждом блоке. Однако если число описанных в блоке идентификаторов невелико, это мало что даст. Если принять, что только внешний блок программы содержит большое число описаний, то можно прибегнуть к компромиссному решению — использовать таблицу хеширования только для идентификаторов, описанных в самом внешнем блоке программы.

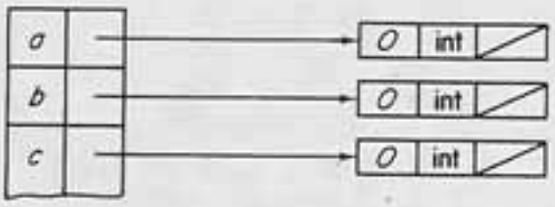


Рис. 8.6

Другой подход заключается в том, что вся таблица символов заменяется таблицей хеширования, в которой каждый элемент является списком, содержащим информацию о типе и т. д. относительно всех описаний конкретного идентификатора в текущем и внешних блоках. Например, после входа в блок 0 в вышеприведенной программе таблица имела бы вид изображенной на рис. 8.6, а после входа в блок 2 — изображенной на рис. 8.7. При появлении в блоке идентификатора, который не был описан в предыдущих блоках, ему выделяется новая запись в таблице хеширования в соответствии с функцией хеширования, а его признаки (номер уровня и тип) вносятся в один элемент списка, соответствующий этой записи. Однако идентификатор, который уже *был* описан в предыдущем блоке (независимо от того, оставлен этот блок или нет), должен иметь запись в таблице. В таком случае необхо-

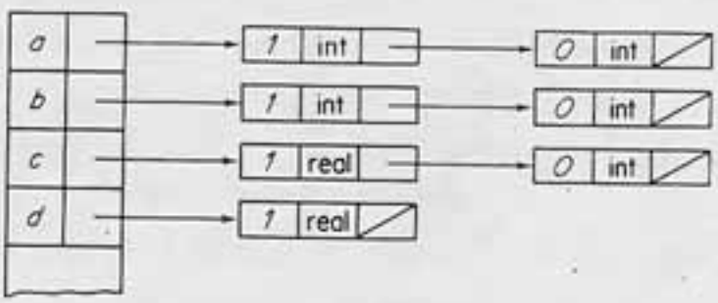


Рис. 8.7

димо лишь добавить в начало списка его признаков (или если у него их нет, поскольку все блоки, содержащие описания этого идентификатора, уже оставлены, генерируется новый элемент списка) те признаки, которые находятся в описании в текущем блоке. При выходе из блока первый элемент каждого списка признаков для каждого идентификатора, описанного в этом блоке, стирается. Зная текущий номер, стирание можно выполнить путем просмотра всей таблицы хеширования или, что еще эффективнее, с помощью стека. С этой целью при входе в блок имена всех описанных в нем идентификаторов помещаются в верхней части стека, а при выходе из блока в записях в таблице хеширования, соответствующих данным идентификаторам, делаются нужные исправления, и идентификаторы удаляются из стека. Заметим, что запись никогда не удаляется из таблицы, иначе функция перехеширования могла бы завершить свою работу слишком рано, найдя пустую запись и прервав процесс поиска. Если идентификатор не был описан ни в одном из текущих блоков, он остается в таблице, но имеет нулевой список признаков. Следовательно, в таблице хеширования должно быть достаточно места для всех идентификаторов, описанных в программе, и даже (для повышения эффективности) немного больше. Проход *A* обеспечивает необходимую информацию, позволяющую правильно определить размер этой таблицы.

В некоторых современных языках, например в языке Ада, объявления идентификаторов и т. п. необязательно должны быть прозрачны для всех внутренних блоков. Это, конечно, сказывается на конструкции таблицы символов, используемой при компиляции. Например, в записи таблицы символов может содержаться дополнительное поле, в котором указывается, для каких блоков идентификатор является прозрачным.

8.2. ТАБЛИЦА ВИДОВ

В таких языках, как Алгол 68, где число видов потенциально бесконечно, нельзя представить какой-либо вид целым числом. Необходимо найти приемлемый (с точки зрения разработчика компилятора) способ представления любого возможного вида. В Алголе 68 имеется, по существу, семь вариантов видов:

1. *Основные виды,* например **int, real, char, bool.**

2. *Длинные и короткие виды,* которые содержат символ **long** или **short**, появляющийся несколько раз перед некоторыми основными видами, например

long long real, short int

3. *Именные виды,* состоящие из символа **ref** и следующего за ним вида, например

ref long int

Значение именного вида можно представить как имя (в терминах Алгола 68) или как адрес значения вида, следующего за **ref.**

4. *Структурные виды,* представленные символом **struct** и последовательностью полей; каждое поле имеет вид и селектор, заключенные в скобки, например

struct (int *i*, **real** *r*, **string** *address*)

Структурные виды соответствуют записям или структурам в других языках.

5. *Виды массивов*, представленные с помощью [ ], [,] или [,,] и т. д. и следующего за этим вида, например

```
[ ]int [,]long real
```

Они используются для выражения значений вида *row* из **int** (одномерного массива) или *row-row* из **long real** (двумерного массива).

6. *Объединенные виды*, состоящие из символа **union** и последовательности представлений вида (описателей), заключенной в скобки, например

```
union (real, int)
```

Эти виды используются для выражения значений, которые могут принадлежать к одному из видов, составляющих данное объединение.

7. *Виды процедур*, представленные символом **proc** и последующим списком видов параметров, заключенным в скобки, за которым следует вид результата, например

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

Эти виды используются для выражения значений, являющихся процедурами.

Теперь выясним, какой тип структуры данных подойдет для представления всех семи вариантов видов. Кроме структурных, объединенных и процедурных, все виды легко представить с помощью последовательности символов, например

```
[ ]long real
ref ref int
ref[ ]ref[ ]bool
```

Для представления вида можно использовать массив или список, причем список более удобен, так как компилятор строит структуру вида слева направо при просмотре программы, и необходимое для представления каждого вида пространство неизвестно, когда встречается его первый символ.

Структурные, объединенные и процедурные виды можно представлять следующим образом. Описатель

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

выражающий значение вида «процедура-с-вещественным-параметром-и-целочисленным-параметром-дающая-логический-результат», может быть представлен структурой с отдельными указателями на список параметров и результат (рис. 8.8).

Аналогичным образом вид

```
struct (int i, struct (int j, bool y), real r)
```

может быть представлен так, как показано на рис. 8.9, а вид

```
union (real, struct (int i, bool b))
```

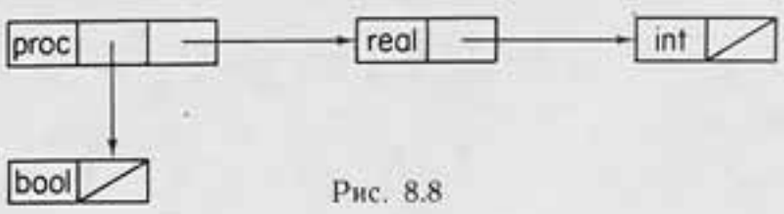


Рис. 8.8

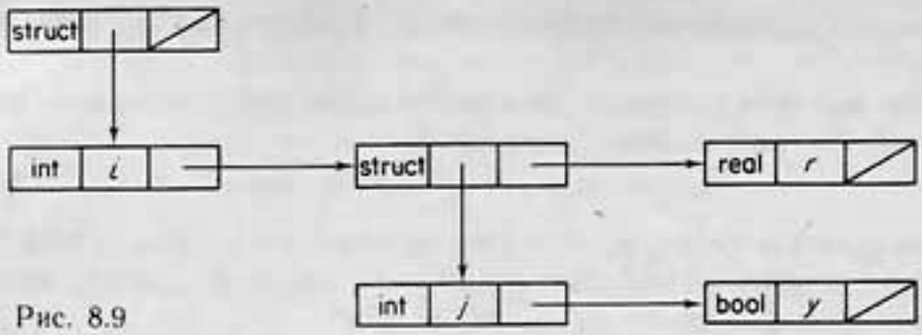


Рис. 8.9

как на рис. 8.10. С помощью такой двумерной структуры можно представить все семь вариантов видов. Вернемся к простому случаю. Вид

ref [ ]ref int

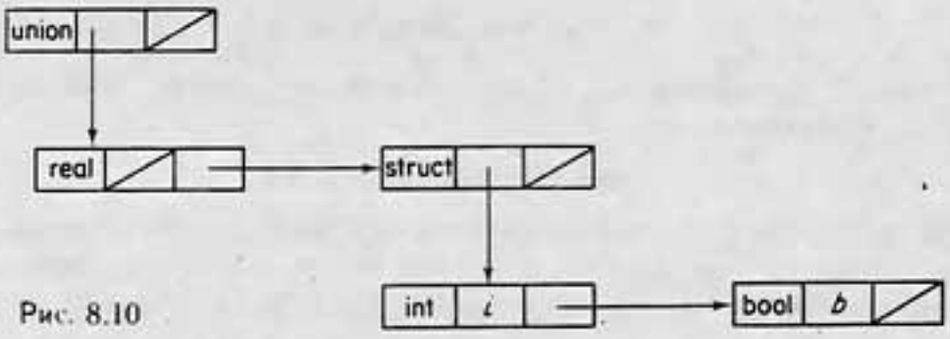


Рис. 8.10

легко представить так, как показано на рис. 8.11.

На практике каждая ячейка имеет два (возможно, пустых) указателя; вертикальный указатель используется в случае структурных, объединенных и процедурных видов.

В данной программе встречается только конечное число видов, и таблица видов содержит ряд указателей на структуры, представляющие различные виды. Информация о видах в таблице символов может в свою очередь состоять из указателей на таблицу видов. С помощью рассмотренного выше метода представления видов компилятор сможет довольно легко (по крайней мере, в смысле требуемых изменений видов) выполнять следующие (в числе других) операции:

1) нахождение вида результата процедуры;

2) выбор поля структуры;

3) разыменование значения, т. е. замену адреса значением в адресе;

4) векторизацию, т. е. преобразование значения кода **int** в **[ ]int** как в

*[1 : 1]***int** *a := 3*

Компилятор должен проверить допустимость всех видов в программе. Точное определение допустимого вида читатель может найти в пересмотренном сообщении об Алголе 68 [55]. Помимо контекстно-свободной грамматики, генерирующей описатели видов, имеются так-

Рис. 8.11

же контекстно-зависимые ограничения. Некоторые из них приводятся ниже.

1. Вид не может быть таким, чтобы его значение потребовало бесконечного объема памяти, например

```
mode a=struct (a first, int second)
```

так что пространством, необходимым для значения вида **a**, служит пространство, необходимое для значения вида **a**, плюс пространство, необходимое для значения вида **int**, и т. д.

2. Вид не может определяться как ряд **ref** и **proc**, за которым следует он сам, например

```
mode b=ref ref b
```

или

```
mode b=ref proc b
```

3. Виды, составляющие объединения, не могут иметь определенных связей, например

```
mode c=union (int, ref int)
```

Допустимый вид обычно считают правильно построенным. Однако поскольку определения видов могут быть взаимно рекурсивными, правильность построения видов большей частью нельзя проверить до окончания прохода, который строит таблицу видов. Например, в

```
mode a=struct (int i, b binf)
mode b=struct (bool b, ref a link)
```

a и **b** являются примерами правильно построенных видов, но проверить это до тех пор, пока не будут встречены оба описания видов, невозможно. Следующие два описания видов представляют недопустимые виды:

```
mode c=(int i, d j)
mode d=long c
```

Из изложенного ясно, что в программе один и тот же вид может быть представлен по-разному. Например, в

```
mode m=ref int;
int a; m b=a
```

a и *b* имеют один и тот же вид. Аналогично виды

```
union (real, int)
```

и

```
union (real, union (real, int))
```

одинаковы. Такое представление вида, как

```
union (int, real)
```

называется написанием. Если два разных написания представляют один и тот же вид, они считаются эквивалентными. Компилятор Алгола 68 должен уметь проверять эквивалентность видов (или, точнее, их написаний). Алгоритм проверки эквивалентности видов приводится в [39].

С таблицей видов, вероятно, должно быть связано несколько проходов компилятора Алгола 68. Во время одного из первых проходов распознаются описатели, так что к его окончанию могут быть полностью распознаны все виды и проверена правильность их построения. При

следующем проходе таблица видов может использоваться для распознавания обозначений операций. Например, в

$$a+b$$

значение + зависит от видов a и b, которые можно определить в таблице видов с помощью таблицы символов. Таблица видов может применяться также в одном из последующих проходов, связанном с комплексными видами, где требуется просматривать структуру данных, представляющую вид.

Упражнения

8.1. Каковы а) преимущества и б) недостатки ограниченной формы, которую могут иметь идентификаторы в Бейсике?

8.2. В Фортране тип идентификатора можно определить по его первой букве. Делает ли это таблицу символов необходимой? Обоснуйте свой ответ.

8.3. Как упростилась бы проблема внесения идентификаторов в таблицу хеширования, если бы не было прохода, во время которого выполняется лексический анализ?

8.4. В некоторых языках имеются такие стандартные идентификаторы, как *SIN* и *COS*. Опишите, каким образом эти идентификаторы могут вноситься
а) в простую таблицу хеширования;
б) в таблицу символов, имеющую блочную структуру.

8.5. Рассмотрите все аргументы «за» и «против» использования указателей в таблице хеширования, как это описано в разд. 8.1.

8.6. Идентификаторы хранятся в алфавитном порядке в одномерном массиве строк. Как построить уравновешенное бинарное дерево, в котором эти идентификаторы являлись бы вершинами, чтобы при его обходе изнутри идентификаторы генерировались бы в алфавитном порядке?

8.7. Какие преимущества с точки зрения пользователя вы видите в том, чтобы иметь возможность употреблять идентификаторы до их описания?

8.8. В таблице видов компилятора Алгола 68 для каждого вида, используемого в программе, должно быть точно одно представление. Обсудите это положение.

8.9. Укажите и обоснуйте, какие из следующих описаний видов являются в Алголе 68 недопустимыми:

(а) **mode x = union (real, ref real)**
(б) **mode y = union (union (real, char), char)**
(в) **mode z = struct (z *z*, int *no*)**

8.10. Правильно ли построены следующие виды для Алгола 68:

mode a = ref ref b
mode b = proc a

Обоснуйте свой ответ.

ГЛАВА 9

РАСПРЕДЕЛЕНИЕ ПАМЯТИ

После выяснения структуры (и значения) программы необходимо выделить место в памяти для значений переменных и т. п. и поместить соответствующие адреса, где это необходимо, в таблицу символов. Фаза распределения памяти почти не зависит от языка и машины и практически одинакова для подавляющего большинства языков, имеющих блочную структуру и реализуемых на многих типичных ЭВМ. Распределение памяти, по существу, заключается в отображении значений, появляющихся в программе, на запоминающем устройстве машины. Если допустить, что мы реализуем типичный язык с блочной структурой, а машина имеет линейное запоминающее устройство, то наиболее подходящим устройством, на котором мы будем базировать распределение памяти, представляется стек или память магазинного типа. В настоящей главе мы рассмотрим классическую структуру стека времени прогона для локального распределения памяти и покажем, как можно произвести глобальное распределение памяти в отдельной области, называемой «кучей».

9.1. СТЕК ВРЕМЕНИ ПРОГОНА

Почти каждой программе требуется какой-либо объем памяти для хранения значений переменных и промежуточных значений выражений. Например, если идентификатор описывается как

int *x*

т. е. *x* может принимать значения (по одному) типа целого, то компилятору придется выделить память для *x*. Иными словами, компилятор должен выделить достаточно места, чтобы записать любое целое число в некотором диапазоне, причем ширина этого диапазона зависит собственно от машины (обычно она определяется целым числом слов или байтов). Аналогично если *y* описывается как

struct (**int** *number*, **real** *size*, **bool** *n*)*y*

то компилятор обеспечивает для значения *y* память с объемом, достаточным для хранения в нем целого, вещественного и логического значения. В обоих этих случаях компилятор не должен испытывать затруднений в вычислении требуемого объема памяти. Точно так же, если *z* описывается как

[*1* : *10*]**int** *z*

объем памяти, необходимой для хранения всех *элементов z*, в 10 раз превышает объем памяти для записи одного целого значения. Однако если бы *w* был описан в виде

[*1* : *n*]**int** *w*

а значение n оказалось бы неизвестным во время компиляции (возможно, оно должно быть считано программой), то компилятор не знал бы, какой объем памяти ему нужно выделить для w. Обычно w называют динамическим массивом. Такие массивы могут описываться в большинстве языков с блочной структурой: в Алголе 60, Алголе 68, ПЛ/1, но не в Паскале. Память для w должна выделяться во время прогона, и на этой стадии требуется, чтобы n имело значение. Память, выделяемая во время компиляции, обычно называется *статической*, а во время прогона — *динамической*. В большинстве компиляторов память для массивов (даже имеющих ограничения констант) выделяется во время прогона, поэтому она считается динамической.

Память нужна также для промежуточных результатов и констант. Например, при вычислении выражения $a+c\times d$ сначала вычисляется $c\times d$, причем значение запоминается в машине, а затем выполняется сложение. Память, используемая для хранения промежуточных результатов, называется *рабочей*. Рабочая память может быть статической или динамической.

В каждом компиляторе предусмотрена схема распределения памяти, которая до некоторой степени зависит от компилируемого языка. В Фортране память, выделенная для значений идентификаторов, никогда не высвобождается, так что здесь подходящей структурой для нее является одномерный массив. Если считается, что массив имеет левую и правую стороны, память можно выделять слева направо. При этом применяется указатель, показывающий первый свободный элемент в массиве. Например, в результате описания

INTEGER A, B, X, Y

выделяется память, как это показано на рис. 9.1. Такая схема не учитывает тот факт, что рабочая память используется неоднократно и весьма неэффективна (в смысле использования объема памяти) для языка с блочной структурой.

В языке, имеющем блочную структуру, память обычно высвобождается при выходе из блока, которому она выделена. В этом случае оптимальным решением было бы разрешить указателю в вышеприведенном примере отодвигаться обратно влево при высвобождении памяти. Такой механизм распределения эквивалентен стеку времени прогона или памяти магазинного типа, хотя принято показывать стек заполняющимся снизу вверх. (Некоторые авторы предпочитают стек, заполняющийся сверху вниз.)

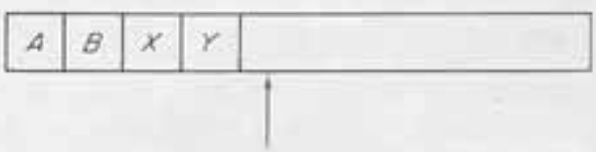


Рис. 9.1

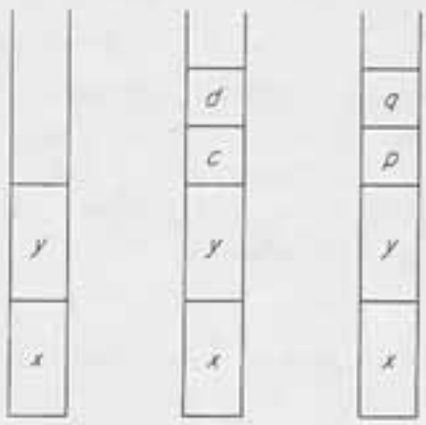


Рис. 9.2

Продолжим описание программы с помощью рис. 9.2, где показаны «моментальные снимки» стека времени прогона на различных этапах ее выполнения.

```
begin real x, y                                    (1)
    .
    .
    .
    begin int c, d                                 (2)
        .
        .
        .
    end;
    begin char p, q                                (3)
        .
        .
        .
    end
        .
        .
        .
end
```

На рис. 9.2 *x*, *y* и т. д. обозначают место, занимаемое значениями этих идентификаторов во время прогона, и мы допустили, что вещественное значение занимает в два раза больше места, чем целое или литерное. Часть стека, соответствующую определенному блоку, называют *рамкой* стека. Считается, что *указатель стека* показывает на его первый свободный элемент.

Кроме указателя стека, требуется также указатель на дно текущей рамки (*указатель рамки*). При входе в блок этот указатель устанавливается равным текущему значению указателя стека. При выходе из блока сначала указатель стека устанавливается равным значению, соответствующему включающему блоку. Указатель рамки включающего блока может храниться в нижней части текущей рамки стека, образуя часть статической цепи или (как мы будем считать) массива, который называется дисплеем. Его можно использовать для хранения во время прогона указателей на начало рамок стека, соответствующих всем текущим блокам (рис. 9.3). Это несколько упрощает перенастройку указателя рамки при выходе из блока. Для возвращения дисплея в исходное состояние при выходе из вызова процедуры можно применять оба эти метода (см. ниже). В любом случае во время прогона здесь выполняется настройка указателей при входе в блок и выходе из него.

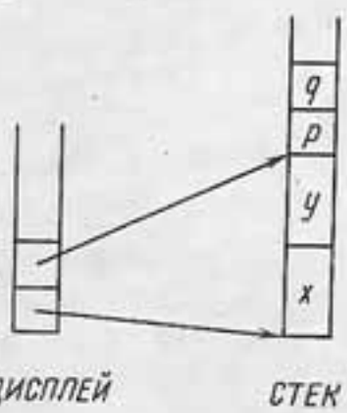


Рис. 9.3

Если бы вся память была статической, адреса времени прогона могли бы распределяться во время компиляции, и значения элементов дисплея также были бы известны во время компиляции.

Рассмотрим следующий отрезок программы:

```
begin int n; read(n);
    [1:n] int numbers;
    real p;
    begin real x,y;
```

Место для *numbers* должно выделяться в первой рамке стека, а для x и y — в рамке над ней. Но во время компиляции неизвестно, где должна начинаться вторая рамка, так как не известен размер чисел. Одно из решений в этой ситуации — иметь два стека: один для статической памяти, распределяемой в процессе компиляции, а другой для динамической памяти, распределяемой в процессе прогона. Однако этого обычно не делают, возможно, из-за тех проблем, которые возникают в связи с наличием более чем одного увеличивающегося и уменьшающегося стека во время прогона. Другое решение заключается в том, чтобы при компиляции выделять статическую память в каждом блоке в начале каждой рамки, а при прогоне — динамическую память над статической в каждой рамке. Это значит, что когда происходит компиляция, мы все еще не знаем, где начинаются рамки (кроме первой), но можем распределять статические адреса *относительно начала определенной рамки*. При прогоне точный размер рамок, соответствующих включающим блокам, известен, так что при входе в блок нужный элемент дисплея всегда можно установить так, чтобы он указывал на начало новой рамки (рис. 9.4).

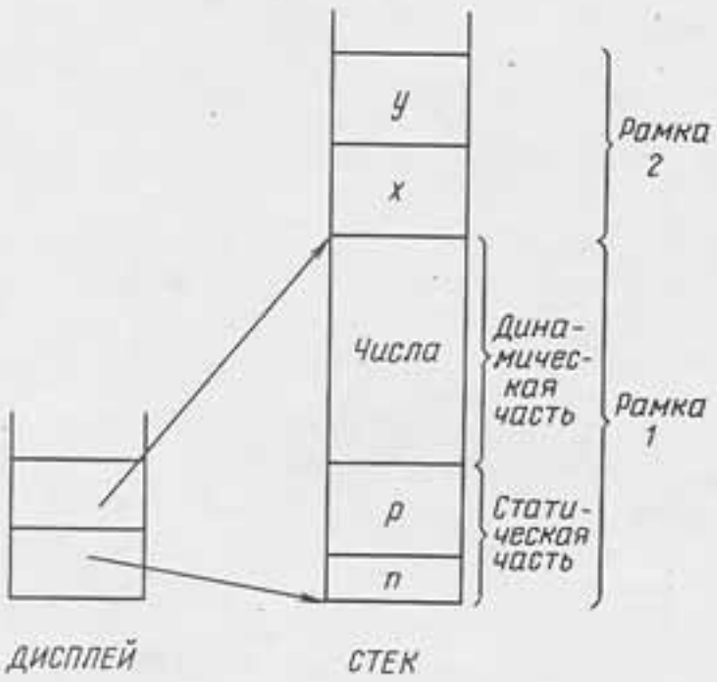


Рис. 9.4

На этом рисунке массив занимает только динамическую память. Однако некоторая информация о массиве обычно *известна* во время компиляции, например его размерность (а следовательно, и число границ — две на каждую размерность), и при выборке определенного элемента массива она может потребоваться. Во многих языках сами

границы могут быть не известны при компиляции, но почти наверняка мы знаем их число, и для *значений* этих границ можно выделить статическую память. Тогда мы вправе считать, что массив состоит из статической и динамической частей. Статическая часть массива может размещаться в статической части рамки, а динамическая — в динамической. Кроме информации о границах, в статической части может храниться указатель на сами элементы массива. Поэтому дополним

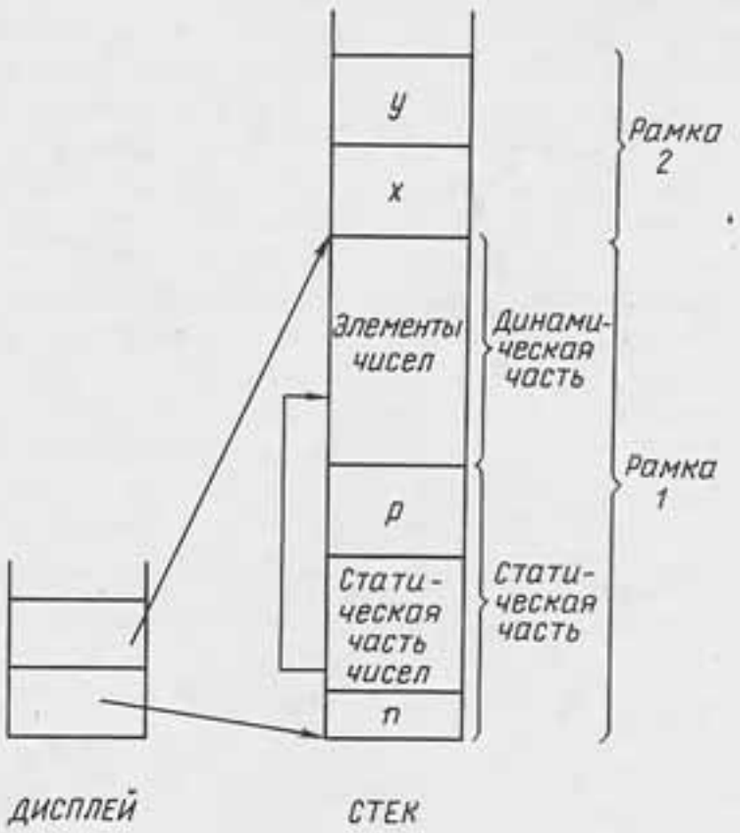


Рис. 9.5

рис. 9.4, показав на нем статическую и динамическую части массива (рис. 9.5).

Когда в программе выбирается конкретный элемент массива, его адрес внутри динамической части должен вычисляться в процессе прогона. Рассмотрим массив

$$[1 : 10, -5 : 5]\,\textbf{int}\ table$$

Будем считать, что элементы записаны в лексикографическом порядке индексов, т. е. элементы таблицы хранятся в следующем порядке:

$$table[1, -5],\ table[1, -4] \ldots\ldots table[1, 5],$$
$$table[2, -5],\ table[2, -4] \ldots\ldots table[2, 5],$$
$$\vdots$$
$$table[10, -5] \ldots\ldots table[10, 5]$$

Адрес конкретного элемента вычисляется как смещение по отношению к базовому адресу (адресу первого элемента) массива:

$$ADDR(table[I,J]) = ADDR(table[l_1, l_2]) + (u_2 - l_2 + 1) \times (I - l_1) + (J - l_2)$$

Здесь l_1 и u_1 — нижняя и верхняя границы первой размерности и т. д. и считается, что каждый элемент массива занимает единицу

объема памяти. Для трехмерного массива соответствующая формула имеет вид

$$ADDR(AR[I,J,K]) = ADDR(AR[l_1, l_2, l_3]) + (u_2-l_2+1)\times(u_3-l_3+1)\times(I-l_1) + (u_3-l_3+1)\times(J-l_2) + K-l_3$$

Выражение (u_i-l_i+1) задает число различных значений, которые может принимать i-й индекс. Например, (u_3-l_3+1) — число различных значений, принимаемых третьим индексом, и, следовательно, в вышеприведенном примере это есть расстояние между элементами массива, отличающимися только на одну единицу во втором индексе.

Аналогично

$$(u_2-l_2+1)\times(u_3-l_3+1)$$

представляет число различных пар значений, которые могут принимать второй и третий индексы, а также расстояние между элементами массива, отличающимися только на одну единицу в первом индексе. Расстояние между элементами, отличающимися на единицу в i-м индексе, известно как *i-й шаг*. Так, в приведенном выше примере первый шаг

$$(u_2-l_2+1)\times(u_3-l_3+1)$$

Он обозначается через s_1. Второй шаг

$$u_3-l_1+1$$

Он обозначается через s_2. Третьим шагом является 1, обозначается он через s_3. На рис. 9.6 показаны шаги для массива

$$[1:5, 1:5, 1:5]\textbf{int } N$$

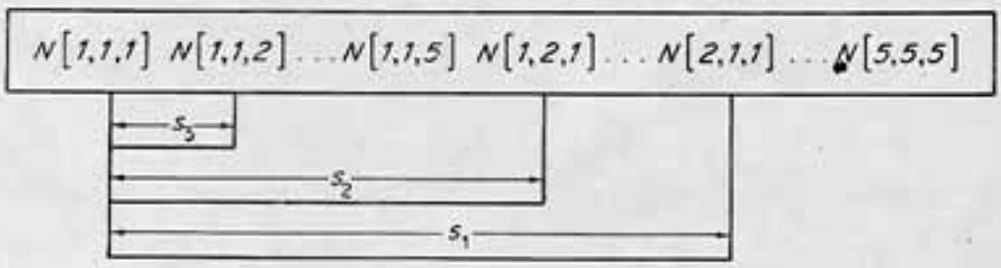


Рис. 9.6

Если бы каждый элемент массива занимал объем памяти r, то эти шаги получали бы умножением всех вышеприведенных величин на r.

Ясно, что вычисление адресов элементов массива в процессе прогона может занимать много времени. Поэтому рекомендуем программистам по возможности избегать выборки из массивов, особенно из многомерных. Тем не менее шаги могут вычисляться только один раз и храниться в статической части массива наряду с границами. Такая статическая информация часто называется описателем массива. В этой же части массива наряду с нижней и верхней границами и шагом для каждой размерности может храниться указатель на элементы массива. Нижняя и верхняя границы требуются для проверки правильности нахождения индексов в пределах границ, а шаги и нижние границы — при обращении к конкретным элементам массива.

Теперь рассмотрим, как распределяется рабочая память при использовании стека. Во многих языках, например в Алголе 60, все идентификаторы должны описываться в блоке, прежде чем можно будет вычислять какие-либо выражения. Отсюда следует, что рабочую

память можно выделять в конкретной рамке стека над памятью, предусмотренной для значений, соответствующих идентификаторам (обычно называемой стеком идентификаторов). Конкретнее статическую рабочую память можно выделять в вершине статического стека идентификаторов, а динамическую рабочую память — в вершине динамического стека идентификаторов. В Алголе 68, где описатели идентификаторов могут появляться после вычисления выражений, стеки идентификаторов и рабочий считают двумя разными стеками во время фазы распределения памяти, хотя для последующих проходов их можно объединять.

В процессе компиляции статический стек идентификаторов растет по мере объявления идентификаторов. Однако статический рабочий стек может не только увеличиваться в размере, но и уменьшаться. Возьмем, например,

$$x := a+b\times c$$

При вычислении выражения $(a+b\times c)$ потребуется рабочая память, чтобы записать значение $b\times c$ перед выполнением сложения. Ту же самую рабочую память можно использовать для хранения результата сложения. Однако после осуществления присвоения этот объем памяти можно освободить, так как он уже не нужен.

Динамическая рабочая память должна распределяться во время прогона, статическая же может распределяться во время компиляции. Объем статической рабочей памяти, который должен выделяться каждой рамке, определяется не рабочей памятью, требуемой в конце каждого блока (обычно она является нулем), а *максимальной* рабочей памятью, требуемой в любой точке внутри блока. Для статической рабочей памяти эту величину можно установить в процессе компиляции, если в фазе распределения памяти мы ассоциируем с рабочей стековой областью текущей рамки два указателя, причем один из них указывает на текущую границу статической рабочей памяти, а другой — на максимальный размер, до которого она выросла при работе с текущим блоком. Именно значение этого второго указателя при выходе из блока и дает объем статического рабочего стека, включаемый в соответствующую рамку.

Поскольку, как уже упоминалось, описания в Алголе 68 необязательно появляются раньше всех вычислений выражений в блоке, статическую рабочую память можно распределять не относительно начала рамки стека, а лишь относительно начала рабочей стековой области этой рамки. Однако в более позднем проходе, когда размер статического стека идентификаторов в какой-либо определенной рамке станет известным, к каждому адресу рабочего стека можно добавить некоторую константу, что сделает его относительным началу рамки стека.

Покажем, как распределяется память в стеке для процедур с тем, чтобы была возможной рекурсия. Обратимся к процедуре

```
proc DIGITS = (int n) int:
begin int m;
  if n < 10 then n
  else m := n ÷ 10;
    n − m × 10 + DIGITS(m)
  fi
end
```

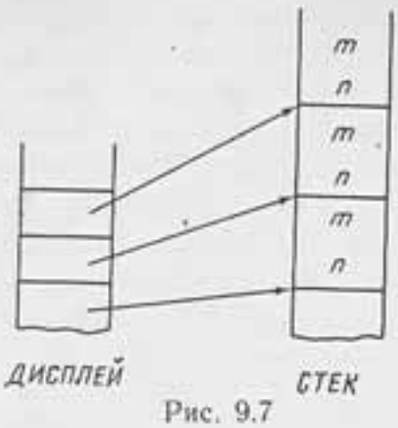


Рис. 9.7

Значение *DIGITS (315)* составит *9*, сумма цифр 3, 1 и 5, а для ее вычисления необходимо три раза входить в эту процедуру. На каждом уровне вхождения выделяется память для *m* и *n*. С этой целью можно вводить новую рамку стека при каждом (рекурсивном) входе в процедуру. Такие рамки называются динамическими (а не статическими). Стек времени прогона после третьего вхождения в процедуру будет выглядеть так, как изображено на рис. 9.7 (стек приведен лишь частично).

Методы вызова параметров

При описании распределения памяти, необходимой для осуществления процедуры *DIGITS*, подразумевалось, что нужно выделять память для формального параметра *n*. Объем выделяемой памяти зависит от метода сообщения между фактическим параметром в вызове процедуры, например *315* в вызове *DIGITS (315)*, и формальным параметром в описании процедуры, *n* в описании *DIGITS*. В различных языках используются различные методы «вызова параметров»; большинство языков предоставляют программисту возможность выбора по меньшей мере из двух методов.

Некоторые методы вызова параметров кратко описываются далее. Под «телом процедуры» мы подразумеваем выполняемую часть процедуры, т. е. ту ее часть, которая заключена между **begin** и **end** в *DIGITS*.

Вызов по значению

Фактический параметр (которым может быть выражение) вычисляется, и копия его значения помещается в память, выделенную для формального параметра. Это — один из методов, предлагаемых Алголом 60 и Паскалем, где формальный параметр может вести себя как локальная переменная и принимать присвоения в пределах тела процедуры. Такое присвоение, однако, не влияет на значение фактического параметра, поэтому данный метод вызова нельзя применять для вывода результата из процедуры. Вызов по значению является эффективным методом передачи информации в процедуру, где большие массивы позволяют делать процесс копирования недорогим.

Вызов по имени

Этот метод применяется в Алголе 60. Он заключается в текстуальной замене формального параметра в теле процедуры фактическим параметром перед выполнением тела процедуры. Там, где фактическим

параметром является выражение, оно должно вычисляться всякий раз, когда в теле процедуры появляется соответствующий формальный параметр. Это — дорогой метод, но при определенных обстоятельствах он может оказаться полезным. В языках, появившихся после Алгола 60, вызов по имени не нашел применения, так как того же результата обычно можно добиться, используя процедуру как параметр процедуры. Такой метод позволяет программисту точнее представить себе связанные с ним затраты. С точки зрения реализации аналогичный результат можно получить с помощью специальной подпрограммы времени прогона для вычисления выражения, соответствующего фактическому параметру, и вызов этой подпрограммы эффективно заменит каждое появление формального параметра в теле процедуры.

Вызов по результату

Как и в вызове по значению, при входе в процедуру выделяется память для значения формального параметра. Однако *никакое* начальное значение формальному параметру не присваивается. Тело процедуры может осуществлять присвоения формальному параметру (и почти наверняка делает это), а при выходе из процедуры значение, которое в этот момент имеет формальный параметр, присваивается фактическому параметру. Этот метод удобен для передачи результата из процедуры и реализован в Алголе W.

Вызов по значению и результату

Этот метод также применяется в Алголе W. Он представляет собой комбинацию вызова по значению и вызова по результату. Копирование происходит при входе в процедуру и при выходе из нее.

Вызов по ссылке

Здесь за адрес формального параметра принимается адрес фактического параметра, если последний не является выражением. В противном случае выражение вычисляется, и его значение помещается по адресу, выделенному для формального параметра. При таком методе мы получаем тот же результат, что и при вызове по значению для выражений или в ином случае вызову по имени. Вызов по ссылке считается хорошим компромиссным решением и осуществлен во многих языках.

В Алголе 68 формальный и фактический параметры делаются эквивалентными с помощью описания тождества. Например, при входе в процедуру в результате вызова

DIGITS(315)

вырабатывается объявление тождества

int *n* = 315

Это очень похоже на вызов по значению, рассмотренный выше, за исключением того, что осуществить присвоение *n* в теле процедуры окажется невозможным, так как его вид будет **int** (а не **ref int**).

В качестве другого примера возьмем процедуру

proc *square* = (**ref int** *a*) **void**:
a := *a*↑2

В результате вызова *square(p)* при входе в процедуру вырабатывается объявление тождества

ref int *a* = *p*

Если считать, что *p* имеет вид **ref int**, то *a* — это просто другой способ написания *p*. Так что в результате мы получим тот же эффект, что и при вызове по ссылке. По этой причине в Алголе 68 применяется только один метод вызова параметров, но он позволяет имитировать большинство из методов, применяемых в других языках.

Обстановка выполнения процедур

Выполнение процедуры в Алголе 68 происходит в обстановке ее описания, а *не* вызова. Рассмотрим, например, следующий отрезок программы:

```
begin int a = 2;
  proc increment = (ref int n) void:
    n plusab a;
  begin int a = 3;
    int p := 4;
    increment (p)
    .
    .
    .
```

Вызов *increment(p)* добавит 2 к *p*, т. е. *a*, появляющаяся в теле процедуры, — это *a*, описанная в начале программы, а *не* во внутреннем блоке. Поэтому при вызове процедуры необходимо внести поправку в стек рабочего времени, чтобы рамка, соответствующая телу процедуры, была немедленно помещена над рамкой, соответствующей блоку, в котором содержится ее описание. Иначе адреса, введенные во время прогона (номер блока, смещение), будут относиться к другой рамке. На практике, чтобы не изменять стек при исключении из него нескольких рамок, можно изменить дисплей, тогда доступ через него дает измененный вид стека. Это изменение должно выполняться сразу же *после* вычисления параметров, так как они вычисляются в языковой обстановке вызова (рис. 9.8).

Конечно, после выхода из процедуры дисплей должен быть восстановлен с помощью статической цепи.

Один из способов осуществления связи между фактическими и формальными параметрами заключается в введении блока дополнитель-

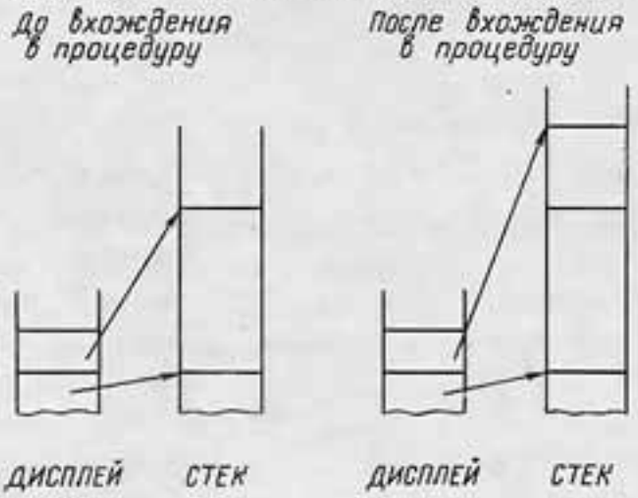


Рис. 9.8

ного уровня (его иногда называют *псевдоблоком*), в котором вычисляются фактические параметры. По завершении их вычисления рамку стека, соответствующую этому блоку, можно использовать в качестве рамки для тела процедуры после видоизменения дисплея. Это позволяет не прибегать к присваиванию параметров.

Очевидно, что входы и выходы из блоков и процедур нам дорого обходятся в смысле времени. Возникает вопрос, нельзя ли как-то сократить время настройки дисплея, особенно когда это касается программ с множеством уровней блоков? Можно не создавать новую рамку для стека каждый раз при входе в новый блок, а, если речь не идет о процедурах (что мы вскоре обсудим), иметь единую рамку для всех значений стека. При этом полностью устранятся все издержки, связанные с входом и выходом из блока, но неперекрещивающиеся блоки не смогут пользоваться одной и той же памятью, и, следовательно, в обмен на сэкономленное время мы получим увеличение объема памяти. Возможно, какой-либо настраивающийся компилятор сможет работать попеременно в двух режимах. В одном режиме он будет оптимизировать скорость выполнения программы (никаких перегрузок при входе в блок и выходе из него), в другом — объем памяти, занимаемой объектной программой во время прогона (обычный метод в отношении входа и выхода из блока). Режим работы компилятора программист может задавать через переключатель. В Алголе 68 это, вероятно, осуществлялось бы с помощью *прагмата*. Если предоставлять программисту такую возможность выбора нежелательно или если эти две меры являются чересчур крайними, разработчик компилятора в тех случаях, когда он предполагает, что недостатки перевесят преимущества, может не создавать новую рамку стека там, где он в обычных условиях это бы сделал. Например, в Алголе 68 необходимость создания новой рамки в цикле

```
for i to 10 do
    print (i)
```

для управляющей переменной *i* представляется спорной. Поскольку других описаний с тем же диапазоном быть не может, ее создание вряд ли можно считать целесообразным.

Что касается процедур, новая рамка стека все же потребуется для каждого (динамического) входа в процедуру, особенно если должна быть осуществлена рекурсия.

Параллельная обработка

Если в реализуемом языке возможна параллельная обработка, т. е. одновременно могут быть активными несколько процессов или процессы могут сливаться каким-либо произвольным способом, стек становится неприемлемым как модель распределения памяти, поскольку объем памяти уже необязательно будет высвобождаться в порядке, обратном тому, в каком он выделяется. (Принцип «первым вошел — первым вышел» уже не применим.) Но можно воспользоваться подходящим обобщением стека, а именно позволить ему иметь ветви, как у дерева, чтобы каждая ветвь соответствовала одному из параллельных

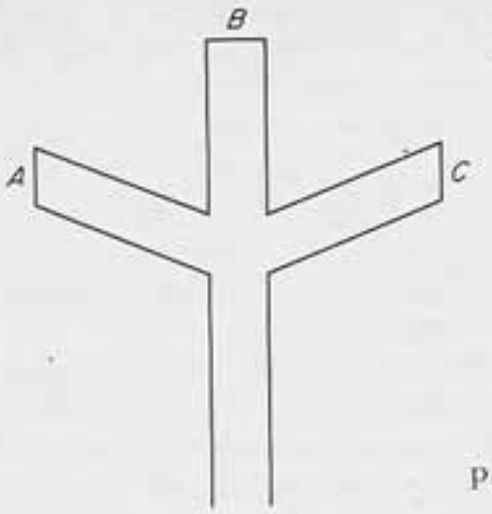


Рис. 9.9

процессов. Например, реализация параллельного предложения в Алголе 68

par (*A*, *B*, *C*)

где *A*, *B* и *C* — параллельно выполняющиеся процессы, потребовала бы распределения памяти, как на рис. 9.9.

Хранение в памяти других значений

Теперь рассмотрим, как хранятся некоторые более сложные значения.

1. *Структуры* в Алголе 68 (записи в некоторых других языках) могут занимать сплошной объем памяти в стеке. Например, значение вида

struct (**int** *i*, **real** *r*, **bool** *b*)

может храниться в стеке просто как целое значение со следующим за ним вещественным значением и следующим логическим значением и занимать тот же объем памяти, какой занимали бы эти три значения, вместе взятые.

2. *Объединения* в Алголе 68 требуют статической памяти, равной максимальной статической памяти, необходимой для любого вида компонента, плюс память для указания, к какому из видов компонента принадлежит данное значение (рис. 9.10). Варианты в Паскале не нуждаются в указателе на фактический вид, так как этого нельзя проверить во время прогона.

3. Для *локальных генераторов* в Алголе 68 память должна выделяться в динамическом стеке, так как требуемый объем памяти не всегда известен во время компиляции (локальный генератор, например, может появиться внутри цикла). Однако каждое статическое появление локального генератора может отмечаться во время компиляции, и в статическом стеке может выделяться память для указателя на его

Рис. 9.10

текущую динамическую реализацию во время прогона. Более подробно организационные методы при прогоне для Алгола 68 рассмотрены в [26].

9.2 КУЧА

До сих пор мы рассматривали только локальную память, удобную для системы распределения памяти, базирующейся на стеке. В большинстве языков программирования обычная блочная структура обеспечивает высвобождение памяти в порядке, обратном тому, в каком она распределялась. Однако в программах со списками и другими структурами данных, содержащими указатели, часто необходимо сохранять память за пределами того блока, в котором она выделялась.

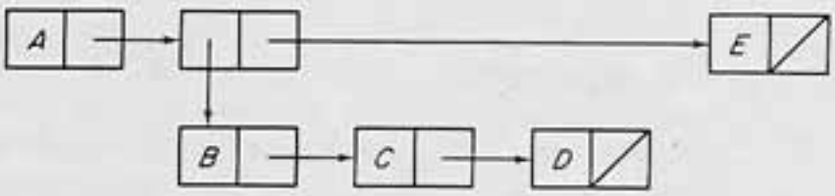


Рис. 9.11

В качестве примера рассмотрим, как организовать список литер в Алголе 68. Подходящим описанием вида служит следующее:

mode list = **struct** (**union** (**char**, **ref list**) *head*, **ref list** *tail*)

Обычный список можно показать схематически, как на рис. 9.11, где *A* представляет литеру «*A*» и т. д. Каждый список состоит из головной части — литеры или указателя на другой список — и хвостовой — указателя на другой список или нулевого списка, представленного на рисунке перечеркнутым квадратом. Этот же список можно записать иначе:

$$(A\,(BCD)\,E)$$

Скобки здесь употребляются для разграничения списков и подсписков.

Для чтения * как данных и организации соответствующего списка можно воспользоваться следующей процедурой:

```
proc readl = ref list:
begin char in;
  read (in);
  read (in);
  if in = ")";
  then nil
  else ref list l = heap list;
  if in = "("
  then read (backspace);
    head of l := readl
  else head of l := in
  fi;
  read (backspace);
  tail of l := readl;
  l
  fi
end
```

read (backspace) в Алголе 68 применяют для того, чтобы вернуть обратно последнюю считанную литеру. Вход в процедуру — рекурсивный для каждой головной и хвостовой части, которая сама является списком. Память для любого элемента списка выделяется глобально (т. е. на время действия всей программы) с помощью генератора, имеющего форму

heap list

Если бы при выходе из процедуры всю память для элементов списка потребовалось перераспределять, то не имело бы смысла применять эту процедуру для его организации. Рассмотрим, однако, следующий фрагмент программы:

ref list *l* := *readl;*
l := **nil**

(*l* — это «указатель» на список). Глобальная память выделяется с помощью *readl,* и после первого присваивания обращение к ней осуществляется посредством идентификатора *l*. Однако после второго присвоения эта память окажется недоступной, так как *l* теперь будет иметь другое значение. Хотя обычно термин «глобальная память» подразумевает, что память выделяется на время действия всей программы, попытка вновь использовать тот объем памяти, который уже становится недоступным программе, представляется вполне разумной. Даже в тех случаях, когда подобная стратегия несколько противоречит духу глобального распределения памяти в таких языках, как Алгол 68, ее целесообразно одобрить, если она не изменяет значения ни одной из программ, т. е. процесс восстановления памяти происходит без участия программиста. Поскольку не существует метода обращения к какому-либо конкретному участку памяти, программист не имеет возможности выяснить, перераспределяется ли эта память для других целей.

Глобальная память может также стать недоступной из-за блочной структуры программы, например в Алголе 68

ref int *c* = **heap int**

генерирует глобальную память для целочисленного значения и ассоциирует имя памяти с идентификатором *c*. Эта ассоциация существует только во время выполнения текущего блока, и если этого не предотвратить, то память, выделенная для значения *c,* станет недоступной, как только текущий блок будет покинут. Одно из возможных решений в такой ситуации — присвоить значение *c* идентификатору вида **ref ref int**, например, *d,* описанному во внешнем блоке

d := *c*

После выхода из блока, в котором описывалось *c*, обращение к этому объему памяти осуществляется (посредством разыменования) через *d*.

Глобальная память не может ориентироваться на стек, поскольку его распределение и перераспределение не соответствуют принципу «последним вошел — первым вышел». Обычно для глобальной памяти выделяется специальный участок памяти, называемый иногда «кучей». Компилятор может выделять память и из стека, и из кучи, и в данном случае уместно сделать так, чтобы эти два участка «росли» навстречу друг другу с противоположных сторон запоминающего устройства (рис. 9.12). Это значит, что память можно выделять из любого участ-

Рис. 9.12

ка до тех пор, пока они не встретятся. Такой метод позволяет лучше использовать имеющийся объем памяти, чем при произвольном ее делении на два участка, ограничивающем и стек, и кучу.

Размер стека увеличивается и уменьшается упорядоченно по мере входа и выхода из блоков. Размер же кучи может только увеличиваться, если не считать тех «дыр», которые могут появляться за счет освобождения отдельных участков памяти. Существуют две разные концепции регулирования кучи. Одна из них основана на так называемых *счетчиках ссылок*, а другая — на *сборке мусора*. Рассмотрим обе эти концепции.

Счетчики ссылок

При использовании счетчиков ссылок память восстанавливается сразу после того, как она оказывается недоступной для программы. Куча рассматривается как последовательность ячеек, в каждой из которых содержится невидимое для программиста поле (счетчик ссылок), где ведется счет числу других ячеек или значений в стеке, указывающих на эту ячейку. Счетчики ссылок обновляются во время выполнения программы, и когда значение конкретного счетчика становится нулем, соответствующий объем памяти можно восстанавливать. Как показывает следующий пример, алгоритм обновления счетчиков ссылок не является тривиальным.

Для простоты допустим, что в каждой ячейке имеются три поля, первое из которых отводится для счетчика ссылок. Это позволит нам показать, как с помощью счетчика ссылок можно регулировать память в виде списка. Эту идею легко распространить и на более сложные структуры. Если X — идентификатор, указывающий на список, его значение может быть таким, как на рис. 9.13:

$Y :=$ *tail* **of** X

Рис. 9.13

Результат присвоения показан на рис. 9.14.

$X := Z$

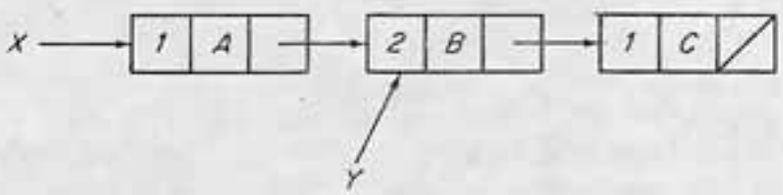


Рис. 9.14

Следующее присвоение может дать результат, приведенный на рис. 9.15.

На единицу уменьшился не только счетчик ссылок ячейки, на которую указывает X, но и ячейка, на которую указывает данная ячейка.

Рис. 9.15

Алгоритм уменьшения счетчика ссылок после присвоения таков: уменьшить на единицу счетчик ссылок ячейки, на которую указывал идентификатор правой части присвоения; если счетчик ссылок является теперь нулем, следовать всем указателям из этой ячейки, уменьшая счетчики ссылок до тех пор, пока (для каждого пути) не будет получено нулевое значение или достигнут конец пути. Поскольку мы считаем, что нельзя следовать параллельно по двум или более путям, нам требуется стек для хранения в нем указателей, которым еще нужно следовать.

Счетчики ссылок в действительности нет необходимости хранить в самих ячейках. Их можно хранить где-нибудь в другом месте, пока соблюдается полное соответствие между адресом каждой ячейки и его счетчиком ссылок.

Метод регулирования памяти в виде кучи с использованием счетчика ссылок имеет два основных недостатка:

1. Память, выделенная для определенных структур, *не* восстанавливается с помощью вышеописанного алгоритма, даже если не будет доступа ни к одному из объемов памяти. Рассмотрим, например, циклический список, показанный на рис. 9.16. Ни один из его счетчиков ссылок не является нулем, хотя никакие указатели извне на него не указывают. Так что этот объем памяти не восстановится.

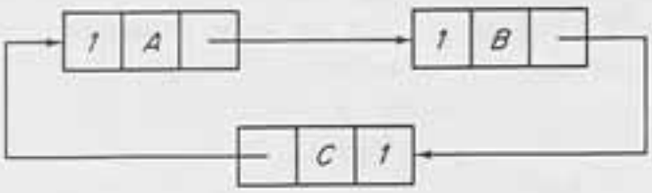


Рис. 9.16

2. Обновление счетчиков ссылок представляет собой значительную нагрузку для всех программ, в том числе и с весьма умеренными потребностями в памяти. Это противоречит принципу Бауэра, гласящему, что «простые программы» не должны расплачиваться за дорогостоящие языковые средства, которыми они не пользуются.

Сборка мусора

Совершенно иной метод высвобождения памяти для ее повторного использования известен под названием *сборки мусора*. Этот метод высвобождает память не тогда, когда она становится недоступной, а только тогда, когда программе требуется память в виде кучи (или, возможно, в виде стека, если эти две области памяти «растут» навстречу друг другу), но ее нет в наличии. Таким образом, у программ с умеренной потребностью в памяти необходимость в ее высвобождении может не возникать. Тем же программам, которым может не хватить объема памяти в виде кучи, придется приостановить свои действия и за-

требовать недоступный объем памяти, а затем уже продолжать свою работу. Конечно, может случиться так, что выполнение программы приостановится для высвобождения памяти, а высвобождать окажется нечего. В этом случае из-за отсутствия памяти программа будет вынуждена завершить свое действие.

Процесс, высвобождающий память, когда выполнение программы приостанавливается, называется сборщиком мусора. В него входят две фазы:

1. *Фаза маркировки.* Все адреса (или ячейки), к которым могут (прямо или косвенно) обращаться идентификаторы, имеющиеся в программе, маркируются либо путем изменения бита в самой ячейке, либо в отображении памяти в другом месте.

2. *Фаза уплотнения.* Все маркированные ячейки передвигаются в один конец кучи (в дальний конец от стека, если эти две области памяти увеличиваются навстречу друг другу).

Фаза уплотнения не тривиальна, так как она может повлечь за собой изменение указателей. Однако в этом разделе мы ограничимся рассмотрением лишь фазы маркировки и приведем некоторые возможные алгоритмы ее выполнения. Конечно, критическим фактором является объем рабочей памяти, имеющийся у сборщика мусора. Будет нелогично, если самому сборщику мусора потребуется большой (не будем говорить уже, что непредсказуемый) объем памяти, поскольку именно недостаток памяти в первую очередь обусловливает вызов сборщика мусора. Конечно, весьма желательно, чтобы сборщик мусора был эффективным по времени, т. е. чтобы программы тратили как можно меньше времени на выполнение (непродуктивной) сборки мусора.

Вне зависимости от процесса редко можно оптимизировать оба фактора сразу — объем памяти и время, и часто приходится принимать какое-то компромиссное решение. Сборка мусора не является исключением из этого правила, и именно выбор таких компромиссных решений делает интересным изучение алгоритмов выполнения фазы маркировки.

Нахождение ячеек, доступных для программы, связано с прохождением по древовидным структурам, так как в ячейках могут содержаться указатели на другие структуры. Необходимо пройти по всем путям, представленным этими указателями, возможно, по очереди, а «очевидное» место хранения указателей, по которым еще придется пройти, будет находиться в стеке. Именно это и входит в функции первого алгоритма, который мы далее опишем. Будем считать для простоты, что каждая ячейка имеет максимум два указателя и поле маркировки, невидимое программисту. Поэтому память представляется массивом структур вида

struct (**int** *left, right*, **bool** *mark*)

Поля *left* и *right* — это целочисленные указатели на другие ячейки; для представления нулевого указателя используется нуль. В данном алгоритме нас интересуют только поля указателей и *mark*, остальные (возможные) поля здесь не показаны.

Алгоритм представлен в виде процедуры *MAR K 1*, где *STAC K* — стек, используемый сборщиком мусора для хранения указателей, *T* — указатель стека, указывающий на его верхний элемент, *A* — массив структур с индексами, начиная с 1, представляющий кучу. После того как все отметки покажут значение «ложь», ячейки, на которые непосредствен-

но ссылаются идентификаторы в программе, или другие значения в стеке времени прогона, маркируются, и их адреса помещаются в *STAC K*. Далее берется верхний элемент из *STAC K*, маркируются непомеченные ячейки, на которые указывает ячейка, находящаяся по этому адресу, и их адреса помещаются в *STAC K*. Выполнение алгоритма завершается, когда *STAC K* становится пустым.

```
proc MARK 1 = void:
begin int k;
  while T ≠ 0
  do k := STACK[T];
    T minusab 1;
    if left of A[k] ≠ 0 and
       not mark of A[left of A[k]]
       then mark of A[left of A[k]] := true;
       T plusab 1;
       STACK[T] := left of A[k]
    fi;
    if right of A[k] ≠ 0 and
       not mark of A[right of A[k]]
       then mark of A[right of A[k]] := true;
       T plusab 1;
       STACK[T] := right of A[k]
    fi
  od
end
```

До вызова этой процедуры куча могла иметь вид табл. 9.1, где маркировано только $A[3]$, потому что на него есть прямая ссылка идентификатора в программе. Результатом выполнения алгоритма будет маркировка $A[5]$, $A[1]$ и $A[2]$ в указанном порядке.

Таблица 9.1

A	*левое*	*правое*	*маркировка*
5	2	1	**false**
4	1	2	**false**
3	5	1	**true**
2	3	0	**false**
1	5	0	**false**

Этот алгоритм очень быстрый (трудно написать более быстрый или более простой алгоритм), но крайне неудовлетворительный, во-первых, потому, что может понадобиться большой объем памяти для стека, во-вторых, потому, что объем требуемой памяти непредсказуем. Следовательно, может отказать сам сборщик мусора из-за нехватки объема памяти для работы.

В другом крайнем случае, когда нужен алгоритм, которому требуется небольшой фиксированный объем памяти, и не так важна эффективность по времени, наилучшим решением представляется процедура *MAR K 2*, нуждающаяся в рабочей памяти только для трех целых чисел: *k*, *k1* и *k2*. Эта процедура просматривает всю кучу в поисках указателей от маркированных ячеек к немаркированным, маркирует последние и запоминает наименьший адрес ячейки, маркированной таким образом. Затем она повторяет этот процесс, но начинает с наименьшего адреса, маркированного в прошлый раз, и так до тех

пор, пока при очередном просмотре не окажется ни одной новой маркированной ячейки. Не исключено, что придется просматривать кучу несколько раз, и если в ней имеется много обратных указателей, алгоритм будет особенно неэффективным.

Как и в предыдущем алгоритме, здесь считается, что ячейки, на которые есть прямые ссылки идентификаторов в программе или значений, находящихся в стеке времени прогона, уже маркированы. Ячейки в куче представлены массивом *A* с границами *1, M*:

```
proc MARK 2 = void:
begin int k, k1, k2;
  k1 := 1;
  while k2 := k1; k1 := M + 1;
  for k from k2 to M
  do if mark of A[k]
     then if left of A[k] ≠ 0 and
       not mark of A[left of A[k]];
     then mark of A[left of A[k]] := true;
       k1 := min (left of A[k], k1)
     fi;
     if right of A[k] ≠ 0 and
       not mark of A[right of A[k]]
     then mark of A[right of A[k]] := true;
       k1 := min (right of A[k], k1)
     fi
     fi
  od;
  k1 ≤ M
  do skip
  od
end
```

min представляет собой процедуру, значением которой является минимум ее двух параметров.

Если этот алгоритм применяется в примере, который рассматривался в связи с предыдущим алгоритмом (*MAR K 1*), то ячейки маркируются в том же порядке (*5, 1, 2*).

Оба эти алгоритма весьма неудовлетворительны (хотя и по разным причинам). Можно, однако, объединить их так, чтобы постараться выявить преимущества (но не недостатки) этих методов. Полученный в результате алгоритм подробно описан Кнутом [35, с. 415]. Суть его заключается в следующем.

Вместо стека произвольного размера, как в *MAR K 1*, применяется стек фиксированного размера. Чем он больше, тем меньше времени может занять выполнение алгоритма. При достаточно большом стеке алгоритм действует точно так же, как *MAR K 1*. Однако поскольку стек нельзя расширять, алгоритм должен уметь обращаться с переполнением, если оно произойдет. Когда стек заполняется, из него удаляется одно значение, чтобы освободить место для другого, добавляемого значения. Для запоминания удаленного таким образом из стека *самого нижнего* адреса используется целочисленная переменная (это аналогично тому, что происходит в *MAR K 2*). Стек работает циклично с двумя указателями: один указывает наверх, другой вниз; это позволяет не перемещать все элементы стека при удалении из него одного элемента (рис. 9.17). Большей частью алгоритм выполняется как *MAR K 1*, и когда стек становится пустым, завершает свою работу, если только

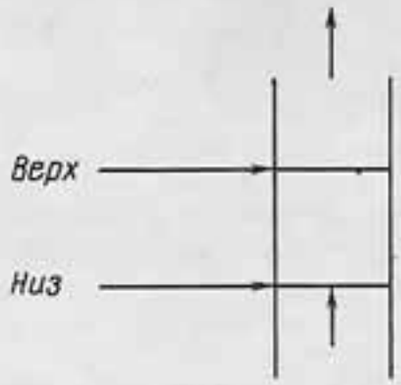


Рис. 9.17

из-за переполнения стека не были удалены какие-либо элементы. Если же элементы удалялись, то узнается самый нижний адрес, который «выпал» из стека, и именно с этого адреса необходимо начать просмотр кучи. Этот процесс аналогичен процедуре *MAR K 2*. Любые другие адреса которые нужно маркировать, помещаются в стек, и, если в конце просмотра он окажется пустым, алгоритм завершается. В противном случае алгоритм вновь ведет себя как *MAR K 1*, и таким образом действует попеременно в обоих режимах до тех пор, пока его выполнение (обязательно) не закончится.

В заключение отметим, что алгоритм (назовем его *MAR K 3*) действует аналогично процедуре *MAR K 1*, когда стек достаточно велик, и работает в смешанном режиме (*MAR K 1* и *MAR K 2*) в противном случае. Это — хороший промежуточный вариант алгоритмов *MAR K 1* и *MAR K 2*, который при достаточно большом стеке будет работать эффективно основную часть времени или, по крайней мере, не откажет.

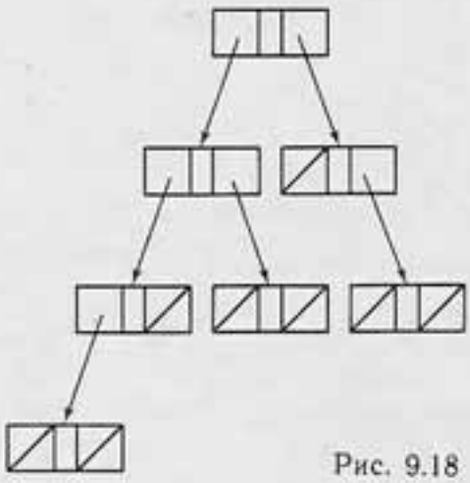


Рис. 9.18

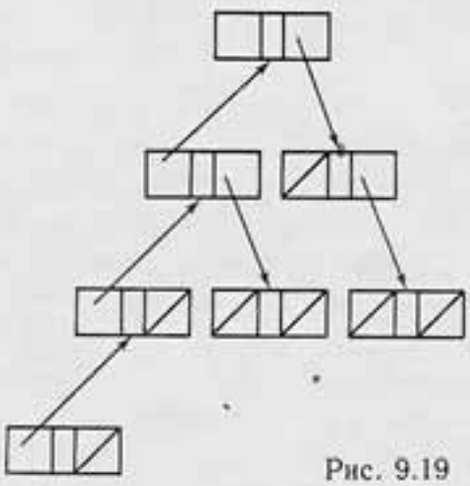


Рис. 9.19

Еще один подход, предложенный Шорром и Уэйтом, также подробно описан Кнутом [35]. Он отличается от предыдущих алгоритмов тем, что в фазе маркировки при сборке мусора структуры, по которым придется проходить, временно изменяются, обеспечивая пути возврата, чтобы можно было обойти все пути. Благодаря этому можно не использовать стек произвольного размера. Допустим, например, что нужно пройти по бинарному дереву, показанному на рис. 9.18. К тому времени, когда фаза маркировки достигнет нижней левой ячейки, это дерево будет выглядеть так, как изображено на рис. 9.19.

Конечно, по завершении маркировки рассматриваемая структура примет свою первоначальную форму. Этому алгоритму требуется одно дополнительное поле, ассоциируемое с каждой ячейкой, в котором долж-

ны учитываться все пройденные пути, и несколько переменных для работы с указателями. Как и в *MARK 1*, время затрачиваемое на выполнение алгоритма, пропорционально числу маркируемых ячеек, но константа этой пропорциональной зависимости будет большей.

Упражнения

9.1. Какое преимущество дает программисту повторное использование памяти в обычном языке, имеющем блочную структуру?

9.2. Чем приходится расплачиваться за возможность обращения к значениям через механизм дисплея? Как этого избежать для статических значений в языке, не имеющем процедур?

9.3. Вместо того чтобы в каждой рамке стека предусматривать объем рабочей памяти, можно использовать отдельный стек. Рассмотрите преимущества и недостатки такого варианта.

9.4. В некотором языке значения определяются глобально или являются локальными по отношению к некоторой процедуре, рекурсивные процедуры не допускаются, и вызовы процедур не могут быть вложенными. Как эти ограничения упростят организацию стека времени прогона?

9.5. В блоке Алгола 60 все описания должны быть сделаны до появления операторов (или выражений). Как это ограничение упростит организацию стека времени прогона?

9.6. Во многих случаях результаты вызова параметра процедуры путем ссылки по значению и по результату одинаковы. Объясните, как можно было бы написать программу, чтобы показать различие этих двух методов.

9.7. В Паскале во время прогона невозможно определить точный вид варианта (как объединения в Алголе 68). Объясните, каким образом это может повлиять на способ хранения значений, являющихся вариантами (по сравнению с объединениями в Алголе 68).

9.8. Рассмотрите, что предпочтительнее использовать в обстановке реального времени — счетчик ссылок или сборку мусора в куче.

9.9. Предложите, как с помощью предварительного просмотра кучи ускорить работу алгоритма *MARK 2*, выполняющего фазу маркировки в сборке мусора.

9.10. Алгоритм Шорра и Уэйта можно применять просто для прохождения по бинарному дереву сверху (см. разд. 10.1). Каковы преимущества этого алгоритма перед обычными методами?

Глава 10

ГЕНЕРАЦИЯ КОДА

Как уже отмечалось, процесс компиляции можно разбить на две стадии — анализ и синтез. Главы 10 и 11 (и в большой степени гл. 9) посвящены стадии синтеза. Проанализировав программу и поместив в таблицы информацию, требуемую для генерации кода, компилятор должен переходить к построению соответствующей программы в машинном коде.

Код генерируется при обходе дерева, построенного анализатором. Обычно генерация кода осуществляется параллельно с построением дерева, но может выполняться и как отдельный проход. Если выполняются два прохода, то представление полного дерева разбора необходимо передать из одного прохода в другой.

В этой и следующей главах мы будем считать, что фактически для получения машинного кода требуются два отдельных прохода:

1) генерация не зависящего от машины промежуточного кода (или объектного языка);

2) генерация машинного кода (или кода сборки) для конкретной машины.

Во многих компиляторах оба эти процесса выполняются за один проход.

10.1. ПРОМЕЖУТОЧНЫЙ КОД

Промежуточные коды (или объектные языки) можно проектировать на различных уровнях. Так, иногда промежуточный код получают, просто разбивая сложные структуры исходного языка на более удобные для обращения элементы. Однако можно в качестве промежуточного кода (в этом случае его чаще называют объектным языком) использовать какой-либо обобщенный машинный код, который затем транслируется в код реальной машины. Получение промежуточного кода возможно до или после распределения памяти. Если это происходит до распределения памяти, то операндами могут служить идентификаторы программы (или их представления после лексического анализа) и присваиваемые компилятором идентификаторы, причем в последнем варианте используются адреса времени прогона.

Одним из видов промежуточного кода являются четверки (см. гл. 6). Например, выражение

$$(-a+b)\times(c+d)$$

можно представить как четверки следующим образом:

$$\begin{aligned} -a &= 1 \\ 1+b &= 2 \\ c+d &= 3 \\ 2\times 3 &= 4 \end{aligned}$$

Здесь целые числа соответствуют идентификаторам, присваиваемым компилятором. Четверки можно считать промежуточным кодом высокого уровня. Такой код часто называют трехадресным — два адреса для операндов (кроме тех случаев, когда имеют место унарные операции) и один для результата. Другой вариант кода — тройки (двухадресный код). Каждая тройка состоит из двух адресов операндов и знака операции. Если сам операнд является тройкой, то используется ее позиция, что исключает необходимость иметь в каждой тройке адрес результата.

Выражение

$$a+b+c\times d$$

можно представить в виде четверок:

$$\begin{array}{l} a+b=1 \\ c\times d=2 \\ 1+2=3 \end{array}$$

и в виде троек:

$$\begin{array}{l} a+b \\ c\times d \\ 1+2 \end{array}$$

Тройки компактнее четверок, но если в компиляторе есть фаза оптимизации, которая пересылает операторы промежуточного кода, их применение затруднительно. Наилучшее решение этой проблемы — косвенные тройки, т. е. операнд, ссылающийся на ранее вычисленную тройку, должен указывать на элемент таблицы указателей на тройки, а не на саму эту тройку.

Как тройки, так и четверки можно распространить не только на выражения, но и на другие конструкции языка. Например, присваивание

$$a := b$$

в виде четверки представляется как

$$a := b=1$$

а в виде тройки — как

$$a := b$$

Аналогично условное предложение

$$\textbf{if } a \textbf{ then } b \textbf{ else } c \textbf{ fi}$$

можно считать выражением с тремя операндами, которому требуются четыре адреса как четверке и три — как тройке.

Не менее популярны в качестве промежуточного кода префиксная и постфиксная нотации. В префиксной нотации каждый знак операции появляется перед своими операндами, а в постфиксной — после. В этом и состоит их отличие от обычной (инфиксной) нотации, в которой обозначения двухместных операций появляются между своими операндами. Например, инфиксное выражение $a+b$ в префиксной нотации примет вид $+\ ab$, а в постфиксной — вид $ab+$.

Префиксная нотация известна также как польская запись (по названию страны ее изобретателя Лукашевича), а постфиксная — как обратная польская запись. С помощью этих нотаций можно записывать более сложные выражения. Например, выражение

$$(a+b)\times(c+d)$$

в префиксной форме записывается следующим образом:

$$\times + ab + cd$$

а в постфиксной так:

$$ab + cd + \times$$

Каждый знак операции в префиксной нотации ставится непосредственно перед своими операндами, а в постфиксной после них.

В префиксной и постфиксной нотациях скобки уже не требуются, так как здесь никогда не возникает сомнений относительно того, какие операнды принадлежат к тем или иным знакам операций. В этих нотациях не существует приоритета знаков операций, хотя при преобразовании инфиксных выражений в префиксные или постфиксные их приоритет, несомненно, нужно учитывать.

Перегруппировку в результате преобразования

$$(a+b)\times(c+d)$$

в

$$ab + cd + \times$$

можно осуществить с помощью стека. Алгоритм такого преобразования хорошо известен (см. [50]). Это преобразование можно выполнить также на основании грамматики инфиксных выражений. В данном случае оно сведется к трем действиям:

1) напечатать идентификатор, когда он встретится при чтении инфиксного выражения слева направо;

2) поместить в стек знак операции, когда он встретится;

3) когда встретится конец выражения (или подвыражения), выдать на печать тот знак операции, который находится в вершине стека.

Этот метод подобен методу, описанному в гл. 6, который применялся для получения четверок, и читателю предлагается закодировать его.

Префиксные и постфиксные выражения можно также получить из представления выражения в виде бинарного дерева. Например, выражение

$$(a+b)\times c+d$$

представляется с помощью бинарного дерева так, как показано на рис. 10.1. Чтобы получить представление префиксного выражения, дерево обходят сверху в порядке, определенном Кнутом [37]:

посещение корня;

обход левого поддерева сверху;

обход правого поддерева сверху,

что дает

$$+ \times + abcd$$

Для получения постфиксного представления дерево обходят снизу. По Кнуту это выглядит так:

обход левого поддерева снизу;

обход правого поддерева снизу;

посещение корня.

В результате имеем:

$$ab + c \times d +$$

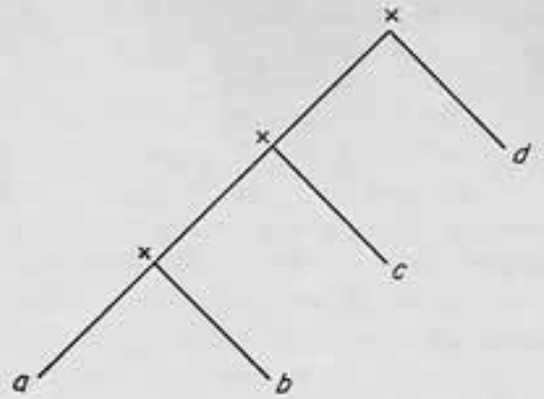


Рис. 10.1

Далее в этой и следующей главах мы будем рассуждать в терминах промежуточного языка (или объектного), состоящего из команд вида

тип-команды параметры

Тип-команды может быть, например, вызовом стандартного обозначения операции, тогда параметрами могут быть имя знака операции, адреса операндов и адрес результата. Например,

STANDOP II+, A, B, C

Здесь *II+* обозначает сложение двух целых чисел, а *A, B, C* служат во время прогона адресами двух операндов и результата. Для того чтобы в промежуточном коде можно было воспользоваться адресами во время прогона, распределение памяти к этому времени должно быть уже закончено. При распределении памяти необходимо знать, какой объем памяти занимает целое, вещественное и другие значения на той машине, для которой выдается объектный код. Это означает, что промежуточный код не является в строгом смысле слова интерфейсом между не зависящей и зависящей от машины частями компилятора. Тем не менее если речь идет о переводе фронтальной части компилятора (т. е. части, транслирующей исходный код в промежуточный) с одной машины на другую, то единственное, что здесь может потребоваться, — это изменение нескольких констант.

Промежуточный код пишется на относительно низком уровне. Он аналогичен коду, использованному [10] для реализации Алгола 68. Обычно выдвигается условие, чтобы промежуточный код отражал структуру реализуемого языка.

Промежуточный код напоминает префиксную нотацию в том смысле, что знак операции всегда предшествует своим операндам. Но он имеет менее общий характер, так как сами операнды не могут быть префиксными выражениями. При получении промежуточного кода для хранения адресов операндов до тех пор, пока не будет напечатан знак операции, используется стек. Поскольку знак операции можно установить (во многих языках) лишь после того, как станут известны его операнды, стек служит также для хранения каждого знака операции на то время, пока не определены оба операнда.

Адрес на время прогона обычно соотносится со стеком, и каждый такой адрес можно представить тройкой вида

(тип-адреса, номер блока, смещение).

Тип-адреса может быть прямым или косвенным (т. е. адрес может

содержать значение или указатель на значение) и ссылаться на рабочий стек или стек идентификаторов. Он может быть также литералом или константой. Номер блока позволяет найти номер уровня блока в таблице блоков (см. разд. 10.2), что обеспечивает доступ к конкретной рамке стека через дисплей. В случае литерала или константы номер блока не используется. Смещение (для адреса стека) показывает смещение значения конкретной рамки по отношению к началу стека идентификаторов или рабочего стека. Если тип-адреса представляет собой литерал, то смещение выражается самим значением, а если тип-адреса — константа, то смещение нужно найти в таблице констант по заданному им адресу. В том случае, когда в каждой рамке стека рабочий стек помещается сразу же над стеком идентификаторов, смещения адресов рабочего стека по отношению к началу рамки можно рассчитывать, как только станет известным размер стека идентификаторов для конкретной рамки (т. е. во время прохода, *следующего* за проходом, при котором происходит распределение памяти).

Адреса во время прогона для идентификаторов определяются в процессе распределения памяти и хранятся в таблице символов вместе с информацией о типе и т. п.

Кроме рассмотренных, существуют и другие команды промежуточного кода (ICI по Бранкару):

SETLABEL LI

для установки метки и

ASSIGN type, add1, add2

для присваивания. Тип необходим как параметр, чтобы определить размер значения, переписываемого из *add1* в *add2*. В Алголе 68 может потребоваться просмотр типа (вида) при трансляции этой команды в фактический код машины, если значения будут содержать динамические части, поэтому во время генерации машинного кода нужна таблица видов.

10.2. СТРУКТУРЫ ДАННЫХ ДЛЯ ГЕНЕРАЦИИ КОДА

В этом разделе мы рассмотрим структуры данных, которые требуются при генерации кода (т. е. генерации промежуточного кода), и их употребление. Как упоминалось выше, для хранения адресов операндов на то время, пока их нельзя будет выдать как параметры ICI, необходим стек значений. В этом стеке, который Бранкар называет *нижним стеком,* можно хранить также и другую информацию. Например, значение может быть связано со своими

а) адресом времени прогона;
б) типом;
в) областью действия,

помимо той информации, которая имеет значение для диагностики. Это — статическая информация, так как (по крайней мере, для большинства языков) ее можно получить во время компиляции. Так, при компиляции может быть известно если не фактическое значение, то во всяком случае адрес целого числа.

При трансляции $A+B$ первыми помещаются в нижний стек статические свойства A. Любой элемент нижнего стека можно представить в виде структуры, имеющей поле для каждой из своих статических характеристик. В случае идентификаторов статические характеристики находятся из таблицы символов. Затем в стек знаков операции поме-

щается знак операции +, и в нижний стек добавляются статические характеристики *B*. Знак операции берется из стека знаков операций, а его два операнда — из нижнего стека. Типы операндов используются для идентификации знака операции, после чего генерируется код. И наконец, в нижний стек помещаются статические характеристики результата.

Этот процесс можно распространить и на более сложные выражения, например, на те, которые генерируются грамматикой с правилами

$$
\begin{aligned}
EXP &\rightarrow TERM \mid \\
&\quad EXP + TERM \mid \\
&\quad EXP - TERM \\
TERM &\rightarrow FACT \mid \\
&\quad TERM \times FACT \mid \\
&\quad TERM / FACT \\
FACT &\rightarrow constant \mid \\
&\quad identifier \mid \\
&\quad (EXP)
\end{aligned}
$$

После чтения идентификатора или константы, знака операции и второго операнда необходимо выполнить следующие действия:

A1. После чтения идентификатора или константы (т. е. листа синтаксического дерева) поместить в нижний стек соответствующие статические характеристики.

A2. После чтения оператора поместить символ операции в стек знаков операций.

A3. После чтения правого операнда (который может быть выражением) извлечь из стеков знак операции и его два операнда, генерировать соответствующий код, так как знак операции идентифицирован, и поместить в стек статические характеристики результата. Тип результата становится известным во время идентификации знака операции, например сложение двух целых чисел всегда дает целое число.

При включении в грамматику этих действий она примет следующий вид:

$$
\begin{aligned}
EXP &\rightarrow TERM \\
&\quad EXP + \langle A2 \rangle TERM \langle A3 \rangle \\
&\quad EXP - \langle A2 \rangle TERM \langle A3 \rangle \\
TERM &\rightarrow FACT \\
&\quad TERM \times \langle A2 \rangle FACT \langle A3 \rangle \\
&\quad TERM / \langle A2 \rangle FACT \langle A3 \rangle \\
FACT &\rightarrow constant \langle A1 \rangle \\
&\quad identifier \langle A1 \rangle \\
&\quad (EXP)
\end{aligned}
$$

Нижний стек частично используется для передачи информации о типе вверх по синтаксическому дереву. Рассмотрим синтаксическое дерево, соответствующее выражению (рис. 10.2):

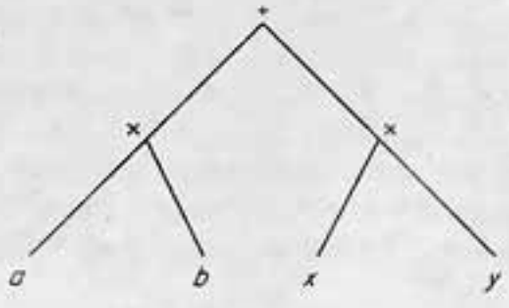


Рис. 10.2

$a \times b + x \times y$

Если значения a и b имеют тип целого, а x и y — тип вещественного значения, компилятор может заключить, воспользовавшись информацией нижнего стека, что «+» в вершине дерева представляет сложение целого и вещественного значений. Мы можем переписать выражение, расставив действия *A1*, *A2* и *A3* в том порядке, в каком они будут вызываться при трансляции этого выражения:

$a\langle A1\rangle\times\langle A2\rangle b\langle A1\rangle\langle A3\rangle+\langle A2\rangle x\langle A1\rangle\times\langle A2\rangle y\langle A1\rangle\langle A3\rangle\langle A3\rangle$

Действие *A3* соответствует применению знака операции. Из изложенного выше вытекает, что каждый вызов *A3* соответствует тому месту, где появился бы знак операции в постфиксной форме. Стек знаков операций, по существу, служит для формирования постфиксной нотации. Поэтому последовательность действий при трансляции данного выражения должна быть следующей:

A1. Поместить статические характеристики a в нижний стек.

A2. Поместить знак «×» в стек знаков операций.

A1. Поместить статические характеристики b в нижний стек.

A3. Извлечь статические характеристики a и b из нижнего стека и знак «×» из стека знаков операций, генерировать код для умножения двух целых чисел, поместить статические характеристики результата в нижний стек; тип результата — целый.

A2. Поместить знак «+» в стек знаков операций.

A1. Поместить статические характеристики x в нижний стек.

A2. Поместить знак «×» в стек знаков операций.

A1. Поместить статические характеристики y в нижний стек.

A3. Извлечь статические характеристики x и y из нижнего стека и знак «×» из стека знаков операций, генерировать код для умножения двух вещественных чисел, поместить статические характеристики результата в нижний стек; тип результата — вещественный.

A3. Извлечь два верхних элемента из нижнего стека и знак «+» из стека знаков операции, генерировать код для сложения целого и вещественного значений, поместить статические характеристики результата в нижний стек; тип результата — вещественный.

Действия *A1*, *A2*, *A3* и вышеприведенную грамматику легко расширить, что позволит использовать

а) большее число уровней приоритета для знаков операций;

б) унарные знаки операций.

Другие случаи употребления нижнего стека рассматриваются в следующем разделе.

Нижний стек обеспечивает передачу информации вверх по синтаксическому дереву. Для передачи же информации вниз по дереву применяется так называемый *верхний стек*. Значение в него помещается всякий раз, когда во время генерации кода происходит вход в такую конструкцию, как присваивание или описание идентификатора. При выходе из этой конструкции значение из стека удаляется. Следовательно, генератор кода может заключить, например, что компилируемое выражение находится справа от знака присваивания; эта информация способствует оптимизации.

Еще одной структурой данных, которая требуется во время генерации кода, является таблица блоков (табл. 10.1). В этой таблице есть запись для каждого блока программы, и эту запись можно рассматривать как структуру, имеющую поля, которые соответствуют номеру уровня блока, размеру статического стека идентификаторов, размеру

Таблица 10.1

Блок	Уровень блока	Размер стека идентификаторов	Размер рабочего стека
1	1	14	16
2	2	12	11
3	2	21	13
4	3	4	9
5	2	6	12

статического рабочего стека и т. д. Такую таблицу можно заполнять во время прохода, генерирующего код, и с ее помощью в следующем проходе вычислять смещения адресов рабочего стека по отношению к текущей рамке стека.

Таким образом, во время генерации кода используются следующие основные структуры данных: нижний стек, верхний стек, стек знаков операций, таблица блоков и, кроме того, таблица видов и таблица символов из предыдущих проходов.

10.3. ГЕНЕРАЦИЯ КОДА ДЛЯ НЕКОТОРЫХ ТИПИЧНЫХ КОНСТРУКТОВ

Покажем, как генерируется код для некоторых конструктов, типичных для языков программирования высокого уровня.

1. *Присваивания*

В соответствии с терминологией Алгола 68 присваивание имеет вид

destination := *source*

Смысл его состоит в том, что значение, соответствующее *источнику*, присваивается значению, которое является адресом (или именем), заданным *получателем*. Например, в

$$p := x+y$$

значение «$x+y$» присваивается p.

Допустим, что статические характеристики источника и получателя уже находятся в вершине нижнего стека. Опишем действия, выполняемые во время компиляции для осуществления присваивания. Прежде всего из нижнего стека удаляются два верхних элемента, после чего происходит следующее:

1. Проверяется непротиворечивость типов получателя и источника. Так как получатель представляет собой адрес, источник должен давать что-нибудь приемлемое для присваивания этому адресу. В зависимости от реализуемого языка типы *получателя* и *источника* можно определенным образом изменять до выполнения присваивания. Например, если тип источника — целое число, то его можно сначала преобразовать в вещественное, а затем присвоить адресу, имеющему тип вещественного числа.

2. Там, где это необходимо, проверяются правила области действия. В Алголе 68 *источник* не может иметь меньшую область действия, чем *получатель*. Например, в

```
begin ref real xx;
  begin real x;
    xx := x
end
```

присваивание недопустимо, и это может быть обнаружено во время компиляции, если в таблице символов или в нижнем стеке имеется информация об области действия. Однако в процессе компиляции нельзя обнаружить все нарушения правил области действия, и в некоторых случаях для проверки этой области приходится создавать код во время прогона.

3. Генерируется код для присваивания, имеющий форму

ASSIGN type, S, D

где S — адрес *источника*, а D — адрес *получателя*.

4. Если язык ориентирован на выражения (т. е. само присвоение имеет значение), статические характеристики этого значения помещаются в нижний стек.

2. *Условные зависимости*

Почти все языки содержат условное выражение или оператор, аналогичный следующему:

if *B* **then** *C* **else** *D* **fi**

При генерации кода для такой условной зависимости во время компиляции выполняются три действия. Грамматика с включенными действиями:

CONDITIONAL→**if** *B*<*A1*>**then** *C*<*A2*>**else** *D*<*A3*>**fi**

Действия *A1*, *A2*, *A3* означают (*next* — значение номера следующей метки, присваиваемое компилятором):

A1. Проверить тип *B*, применяя любые необходимые преобразования (приведения) типа для получения логического значения. Выдать код для перехода к *L*<*next*>, если *B* есть «ложь»:

JUMPF L<*next*>,<*address of B*>

Поместить в стек значение *next* (обычно для этого служит стек знаков операций). Увеличить *next* на 1. (Угловые скобки (<, >), в которые заключаются «*next*» и «*address of B*», используются для обозначения значений этих величин, и их не следует путать со скобками, в которые заключаются действия в порождающих правилах грамматики.)

A2. Генерировать код для перехода через ветвь **else** (т. е. перехода к концу условной зависимости)

GOTO L<*next*>

Удалить из стека номер метки (помещенный в стек действием *A1*), назвать *i*, генерировать код для размещения метки

SETLABEL L<*i*>

Поместить в стек значение *next*. Увеличить *next* на 1.

A3. Удалить из стека номер метки (*j*). Генерировать код для размещения метки

SETLABEL L<j>

Если условная зависимость сама является выражением (а не оператором), компилятор должен знать, где хранить его значение, независимо от того, какая часть вычисляется — **then** или **else**. Это можно сделать, специфицируя адрес, который указывает на данное значение, или пересылая значение, заданное частью **then** либо частью **else**, по указанному адресу.

Аналогично можно обращаться с вложенными условными выражениями или операторами.

3. *Описания идентификаторов*

Допустим, что типы всех идентификаторов полностью выяснены в предыдущем проходе и помещены в таблицу символов. Адреса распределяются во время прохода, генерирующего код.

Рассмотрим описание

somemode *x*

Перечислим действия, выполняемые во время компиляции:

1. В таблице символов производится поиск записи, соответствующей *x*.

2. Текущее значение указателя стека идентификаторов дает адрес, который нужно выделить для *x*. Этот адрес

(idstack, current block number, idstack pointer)

включается в таблицу символов, а указатель стека идентификаторов увеличивается на статический размер значения, соответствующего *x*. (В Алголе 68, если вид *x* начинается с **ref**, объем памяти должен выделяться для значения, на которое ссылается *x*, а не для самого *x*; это обозначается с помощью адреса другого типа.)

3. Если *x* имеет динамическую часть, например в случае массива, то генерируется код для размещения динамической памяти во время прогона.

4. *Циклы*

Рассмотрим следующий простой цикл:

for *i* **to** *10* **do** *something* **od**

Для генерации кода требуются четыре действия, которые размещаются следующим образом:

for *i*<*A1*>**to***10*<*A2*>**do**<*A3*>*something*<*A4*>**od**

Эти действия таковы:

A1. Выделить память для управляющей переменной *i*. Поместить сначала в эту память *1*

MOVE «1», address (controlled variable)

A2. Генерировать код для записи в память значения верхнего предела рабочего стека

MOVE address (ulimit), (wostack, current block number, wostack pointer)

(*wostack pointer* — указатель рабочего стека). Увеличить указатель

рабочего стека и уменьшить указатель нижнего стека, где хранились статические характеристики верхнего предела.

A3. Поместить метку

SETLABEL L<next>

Увеличить *next* на 1.

Выдать код для сравнения управляющей переменной с верхним пределом и перейти к *L<next>*, если управляющая переменная больше верхнего предела:

JUMPG L<next>, address (controlled variable), address (ulimit)

Поместить в стек значение *next*. Поместить в стек значение *next* — 1. Увеличить *next* на *1*.

A4. Генерировать код для увеличения управляющей переменной

PLUS address (controlled variable), 1

Удалить из стека номер (*i*). Генерировать код для перехода к *L<i>*

GOTO L<i>

Удалить из стека номер метки (*j*). Поместить метку в конец цикла

SETLABEL <j>

Цикл

for *i* **to** *10* **do** *something* **od**

генерирует код следующего вида:

```
MOVE «1», address (controlled variable)
MOVE address (ulimit), wostack pointer
SETLABEL L1
JUMPG L2 address (controlled variable), address (ulimit)
(something)
PLUS address (controlled variable), 1
GOTO L1
SETLABEL L2
```

Действие *A4* можно видоизменить, если приращение управляющей переменной будет не стандартным, равным 1, а иным, например в

for *i* **by** *5* **to** *10* **do**

Для этого, возможно, придется вычислять выражение и хранить его значение в рабочем стеке, чтобы использовать как приращение. В Алголе 68 приращение вычисляется один раз перед входом в цикл, после чего его нельзя изменить.

Аналогично в

for *i* **from** $m+n$ **to** *20* **do**

начальное значение управляющей переменной может быть выражением, значение которого хранилось в рабочем стеке. Можно также включить случай с отрицательным приращением управляющей переменной.

Если в цикле содержится часть **while**, как, например, в

for *i* **to** *10* **while** $a<b$ **do**

действие *A3* следует видоизменить, чтобы при принятии решения о выходе из цикла учитывались значения как части **while**, так и управляющей переменной, причем любая из этих проверок достаточна для завершения цикла.

5. *Вход и выход из блока*

При входе в блок (последовательное предложение с описаниями в Алголе 68) предположим, что во время предыдущего прохода получена определенная информация. Она состоит из таблиц видов и символов, дающих типы или виды всех идентификаторов и т. п. Тогда при входе в блок нужно выполнить следующие основные действия:

1. Прочитать в таблице символов информацию, касающуюся блока, и связать ее с информацией включающих блоков таким образом, чтобы можно было выполнить «внешние» поиски определяющих реализаций идентификаторов и т. д. (см. разд. 8.1).

2. Поместить в стек (*idstack pointer*). Поместить в стек (*wostack pointer*). Поместить в стек (*block number*). Все эти значения ссылаются на включающий блок и могут потребоваться вновь после того, как будет покинут блок, в который только что осуществлен вход:

idstack pointer := *0*
wostack pointer := *0*

3. Генерировать код для исправления *DISPLAY*

BLOCK ENTRY block number

4. Увеличить номер уровня блока на 1. Увеличить *gbn* (наибольший использованный до сих пор номер блока) на 1 и присвоить это значение номеру блока.

5. Прочитать информацию о виде и добавить ее в таблицу видов (если в языке имеются такие «сложные» виды, как в Алголе 68).

При выходе из блока требуется:

1. Обновить таблицу блоков, задав размер стека идентификаторов и т. п. для покинутого блока.

2. Исключить информацию в виде таблицы символов для покинутого блока (см. разд. 8.1).

3. Генерировать код для изменения дисплея

BLOCK EXIT block number

4. Удалить из стека (*block number*). Удалить из стека (*wostack pointer*). Удалить из стека (*idstack pointer*). Уменьшить номер уровня блока на 1.

5. Поместить результат (при необходимости) в рамку стека вызывающего блока.

6. *Прикладные реализации*

Во время компиляции в соответствии с прикладной реализацией идентификатора, например *x* в

$$x+4$$

необходимо:

1. Найти в таблице символов запись, соответствующую определяющей реализации, например **int** *x* идентификатора.

2. Поместить в нижний стек статические характеристики значения, соответствующего идентификатору.

Подразумевается, что в нижний стек также помещаются стати-

ческие характеристики других терминалов (в грамматике), таких, как константы и т. п.

10.4. ПРОБЛЕМЫ, СВЯЗАННЫЕ С ТИПАМИ

При обсуждении работы генератора кода с определенными типичными конструкциями языков программирования высокого уровня мы почти не касались проблем, связанных с типами, которые могут быть относительно простыми, как в Фортране или Бейсике, и довольно сложными, как в Алголе 68 или ПЛ/1. В компиляторе Алгола 68 более половины кода может приходиться на обращение с видами.

Кто-то однажды сказал, что беда Алгола 68 состоит в том, что приходится изучать все приведения, какие только могут иметь место. Возможно, это правильно, но вывод о наличии большого числа приведений (или автоматических изменений видов) по меньшей мере неверный. По сравнению с ПЛ/1, где автоматические преобразования вида, такие, как целый в литерный или литерный в вещественный, происходят почти в любом контексте, диапазон приведений в Алголе 68 крайне ограничен. В нем возможны шесть приведений:

1. Распроцедуривание, например переход от вида **proc real** к виду **real.**
2. Разыменование, например переход от вида **ref real** к виду **real.**
3. Объединение, например переход от вида **real** к виду **union (real, char)**.
4. Векторизация, например переход от вида **real** к виду **[ ] real.**
5. Обобщение, например переход от вида **int** к виду **real.**
6. Чистка, например переход от вида **int** к виду **void.**

Возможность осуществления приведения зависит от синтаксической позиции. Например, в левой части присваивания может иметь место только распроцедуривание (вызов процедуры без параметров), а в правой части — любое из шести приведений. Иногда бывает необходимо производить несколько приведений. Например, если x имеет вид **ref real** и a — **ref int,** то прежде чем произойдет присваивание

$$x := a,$$

a необходимо сначала разыменовать, а затем обобщить.

В зависимости от того, какие приведения могут выполняться в синтаксических позициях, последние называют мягкими, слабыми, раскрытыми, крепкими или сильными. Например, левая часть присваивания является мягкой (допускается только распроцедуривание), а правая часть — сильной (допускается любое приведение). Кроме ограничения типов приведений, разрешаемых в заданной синтаксической позиции, существуют правила, определяющие порядок осуществления различных приведений. Например, объединение может произойти только один раз и не должно следовать за векторизацией. Можно определить грамматику, которая генерирует все допустимые последовательности приведений в заданной синтаксической позиции, например

SOFT→*deprocedure* |
deprocedure SOFT

Любое предложение, генерированное посредством *SOFT*, представляет собой допустимую последовательность приведений в мягкой синтаксической позиции (т. е. в левой части присваивания).

Для раскрытой позиции (например, индекс в *A* [*i*]) справедливы следующие правила:

MEEK→*deprocedure* |
deprocedure MEEK |
dereference |
dereference MEEK

Другими словами, в раскрытой позиции можно выполнять распроцедуривание и разыменование любое число раз и в любом порядке, например из вида

ref proc ref int

в вид **int.**

Для сильной позиции (например, правая часть присваивания или параметр в вызове процедуры) правила таковы:

STRONG→*dereference STRONG* |
deprocedure STRONG |
unite |
unite ROW |
widen |
widen widen |
widen ROW |
widen widen ROW |
ROW
ROW→*row* |
row ROW

Чистка на этой стадии игнорируется.

Вид до выполнения приведений называется априорным, а после их выполнения — апостериорным. В случае сильных и раскрытых синтаксических позиций известны и априорный, и апостериорный виды. Для других позиций известен лишь априорный вид и некоторая информация об апостериорном виде, например, о том, что он должен начинаться со **struct** или **ref struct,** как в выборе, или о том, что он *не* должен начинаться с **proc,** как в левой части присваивания. Компилятор, применяя соответствующую грамматику, генерирует последовательность приведений из априорного вида либо к известному, либо к подходящему апостериорному виду. Если нельзя найти никакой последовательности приведений, программа синтаксически неправильна. С другой стороны, если подходящая последовательность существует, компилятор применяет приведения по порядку, генерируя, очевидно, в ходе этого процесса код времени прогона.

Чистка представляет собой особую форму приведения в том смысле, что она, по существу, отбрасывает вид (и значение). Чистка происходит, например, в тех местах, где стоит точка с запятой, как в

x := *y*;

Характер конструкции, которая дает очищенное значение, определяет, является ли эта чистка единственным выполняемым приведением или ей должны предшествовать другие приведения. Так, в случае присваивания

p := *q*

значение *p* просто теряется, и никакие другие приведения не осуществляются *независимо от того, какой вид p имеет*. Однако в

;*p*;

где *p* есть прикладная реализация идентификатора и вид *p* начинается с **proc** или **ref proc** и т. д., *p* нужно сначала распроцедурить или разыменовать и распроцедурить, а затем очистить. Обычно именно этого и следует ожидать, причем когда происходит чистка, компилятор должен знать тип той конструкции, которая поставляет данное значение, а также вид этого значения. Более подробно чистка описывается в [43].

Еще одна сложность связана с выбирающими предложениями. В предложении

x+**if** *b* **then** *1* **else** *2.3* **fi**

во время компиляции необходимо знать вид правого операнда знака «+». Другими словами, все варианты выбирающего предложения должны приводиться к общему виду, называемому объектным. Этот процесс известен как уравнивание, и его правила подразумевают, что последовательности сильных приведений можно применять во всех вариантах, кроме одного, в котором используются лишь последовательности приведений, уместные для синтаксической позиции выбирающего предложения. В вышеприведенном примере выбирающее предложение находится в крепкой синтаксической позиции, которая не допускает расширения. Однако внутри выбирающего предложения один вариант допускает сильное приведение, что может повлечь за собой расширение. В этом случае объектным видом окажется **real**, и первый вариант следует расширить, а второй (который, по сути, находится в крепкой позиции) нежелательно подвергать приведению. Алгоритм определения объектного вида в выбирающем предложении описывается в [38].

Действие компилятора при обращении с выбирающими предложениями заключается в том, что статические характеристики всех вариантов выбирающего предложения помещаются в нижний стек, а затем выводятся объектный вид и различные последовательности приведений для каждого варианта. Если какая-либо из этих последовательностей вызывает необходимость генерации кода во время прогона, ее можно выделить в отдельный поток, и между этими двумя потоками ввести указатели, чтобы во время последующего прохода код можно было соединить в нужном порядке.

10.5. ВРЕМЯ КОМПИЛЯЦИИ И ВРЕМЯ ПРОГОНА

Как было показано ранее на примерах, генератор кода во время компиляции выполняет некоторые действия (обращается к нижнему стеку и т. д.), а также генерирует код для других операций, которые будут выполняться во время прогона. Что именно должно быть сделано в процессе компиляции, а что — в процессе прогона, зависит в какой-то степени от компилируемого языка. Например, в POP-2, где типы идентификаторов являются динамическими (т. е. могут динамически изменяться при прогоне), их проверку необходимо осуществлять во время прогона, и для выполнения такой проверки должен генерироваться код. В Алголе 68 все виды — статические (т. е. известны во время компиляции), и их проверку можно выполнять во время компиляции.

Тем не менее, хотя язык и влияет на решение вопроса о том, что осуществлять во время компиляции, а что — во время прогона, разработчик компилятора имеет некоторую свободу действий в этом плане.

Например, в Алголе 68 не вполне ясно, повлечет ли разыменование за собой какое-либо действие при прогоне или нет. На первый взгляд может показаться, что нет, так как при этом просто происходит замена в памяти машины одного значения другим, и изменение адреса времени прогона в процессе компиляции должно быть возможным. Новым адресом будет тот, на который указывает первоначальный адрес, и несмотря на то, что значение указателя может быть неизвестным при компиляции, новый адрес можно обозначить, изменив тип первоначального адреса так, чтобы в нем указывался дополнительный косвенный уровень. Но может ли значение разыменовываться во время компиляции произвольное число раз без генерации кода? Если может, то в наших типах адресов мы должны указывать произвольное число косвенных уровней, что практически едва ли вероятно. Следовательно, в некоторых случаях для осуществления разыменования во время прогона требуется выполнить действие (переслать значение по новому адресу).

Для повышения эффективности выдаваемого кода при компиляции можно проделать дополнительную работу, которую обычно называют оптимизацией. В оптимизацию входит удаление кода из циклов там, где это не влияет на значение программы, исключение возможности вычисления идентичных выражений более чем один раз и т. д. Описание некоторых известных методов оптимизации можно найти у Гриса [20]. Кроме того, в Алголе 68, например, если провести тщательный анализ во время компиляции, можно иногда избежать перезаписи сложных структур данных во время прогона [52].

Компиляторы, выполняющие обширную оптимизацию, компилируют программы дольше, чем те, которые ее не выполняют. Решение вопроса о включении оптимизации зависит от целей проектирования компиляторов. По сообщению Хорнинга [29], проведение некоторых локальных видов оптимизации (например, «узких» — см. разд. 11.3) не требует больших затрат, так что их почти всегда имеет смысл включать. Большинство же глобальных методов оптимизации уместно лишь в тех случаях, когда требование эффективности во время прогона приобретает первостепенную важность.

Упражнения

10.1. Представьте следующее выражение:
а) в тройках;
б) в четверках

$$(a+b+c)\times(d+e+f)$$

10.2. Почему употребляются косвенные тройки?

10.3. Какие проблемы возникают при преобразовании выражений, в которых используются унарные знаки операций, в обратную польскую запись?

10.4. В некоторых первых компиляторах выражения до генерации кода преобразовывались так, чтобы быть полностью заключенными в скобки. Например,

$$a+b\times c$$

заменялось на

$$(a+(b\times c))$$

Каковы преимущества и недостатки этого метода?

10.5. В некоторых языках все знаки операций обладают одинаковым приоритетом, но вычисляются они строго слева направо, т. е.

$$a+b\times c$$

вычисляется как

$$(a+b)\times c$$

Прокомментируйте этот прием с точки зрения реализации.

10.6. В АПЛ все знаки операций имеют одинаковый приоритет, но выражения вычисляются справа налево, т. е.

$$a+b\times c$$

вычисляется как

$$a+(b\times c)$$

Как это влияет на реализацию?

10.7. В некоторых ранних компиляторах тип каждого идентификатора включался в текст рядом с самим идентификатором. Каковы преимущества и недостатки такого метода?

10.8. Опишите, как может быть реализован оператор *case* или *switch* в том языке, с которым вы знакомы.

10.9. Опишите, как может быть реализован оператор *goto*.

10.10. Прокомментируйте с точки зрения эффективности вид команд промежуточного кода, рассмотренных в данной главе.

Глава 11

ГЕНЕРАЦИЯ МАШИННОГО КОДА

Эту главу мы посвящаем конечной задаче компилятора — выработке машинного кода для конкретной машины. Для простоты будем считать, что вырабатывается код Ассемблера (что вполне реально). Допустим, промежуточный код (или объектный код), из которого вырабатывается код Ассемблера, не зависит от машины (но может зависеть от языка). В целом любая команда промежуточного кода влечет за собой появление последовательности команд Ассемблера, в том числе и вызовы подпрограмм Ассемблера. Трансляция каждой команды промежуточного кода предполагается не зависящей от трансляции предшествующей или последующей команды промежуточного кода, за исключением, возможно, зависимости, связанной с ее влиянием на глобальные переменные времени компиляции.

Выработку кода из промежуточного мы иногда далее называем трансляцией, чтобы отличать ее от процесса генерации промежуточного кода, который мы называем генерацией кода.

11.1. ОБЩИЕ ПОЛОЖЕНИЯ

Транслятор состоит из последовательности глобальных описаний процедур и т. п. и следующего за ней цикла, включающего вариантное предложение с элементом для каждого типа команды промежуточного кода. Этот цикл имеет вид:

```
global declarations
  do case read (n); n
    in. . .
    esac
  od
```

При этом допускается, что все типы команд промежуточного кода представлены целыми числами. (Можно использовать строку, но это менее эффективно, так как тогда для выбора соответствующего элемента вариантного предложения придется производить поиск в таблице.) Трансляция завершается считыванием специального значения *n* и выходом из цикла.

Для каждого элемента вариантного предложения (кроме завершающего работу транслятора) требуется чтение параметров команды промежуточного кода. Эти параметры обычно делятся на две группы:

1) целые числа, соответствующие типам (возможно, указатели на таблицу видов);

2) адреса во время прогона: тип-адреса, номер блока, смещение.

Виды могут включаться в таблицу во время синтаксического анализа из таких описаний, как **int** *j* или даже **amode** *k*, где **amode** опре-

деляется пользователем. Однако виды могут также включаться во время генерации кода, где они выдаются как результаты операций или с помощью приведений в Алголе 68. Таблица, используемая транслятором, может содержать не все виды, встречаемые в программе, а только те, которые появляются в качестве параметров в промежуточном коде.

Номер блока в адресах времени прогона (там, где он есть) преобразуется в номер уровня блока с помощью таблицы блоков, что дает возможность обращаться к соответствующему уровню дисплея. При трансляции размеры каждой области различных рамок стека, как правило, известны. Это позволяет соотнести поля смещений адресов, которые могут, например, ссылаться на области рабочего стека рамки, с началом этой рамки. Некоторые типы-адресов на этой стадии исчезают, например те, которые требуют, чтобы смещения соотносились с рабочим стеком. Тем не менее все же придется различать прямые и косвенные адреса. В тех случаях, когда машина не позволяет осуществлять косвенную адресацию, адреса предстоит имитировать с помощью индексных регистров. Например, в ICL 1900

LDX 2 0(1)

(*0(1)* означает адрес *0*, модифицированный значением в регистре *1*), вызывает загрузку содержимого адреса, на который указывает регистр *1*, в регистр *2*. Если адресом служит литерал, смещение не нужно преобразовывать, а если затрагивается таблица констант, может быть выбран соответствующий ее элемент. На этой стадии компилятор должен проверять, имеют ли элементы таблицы констант необходимую для машины (или реализации) длину. При чрезмерно длинных константах или других отклонениях транслятор выдает сообщение об ошибке. Это — единственное сообщение об ошибке, которое выдается транслятором.

Трансляция команды промежуточного кода заключается в чтении параметров, преобразовании адресов и генерации соответствующего кода.

Для преобразования адресов можно воспользоваться стандартной процедурой, которая сама генерирует код, что позволяет нам применить косвенный метод. В результате вызова этой процедуры

convert (add1, 1)

регистр *1* будет указывать на значение в *add1*.

11.2. ПРИМЕРЫ ГЕНЕРАЦИИ МАШИННОГО КОДА

С большинством команд промежуточного кода, трансляция которых здесь описывается, мы уже встречались в разд. 10.3. Команды выдаются в коде *PLAN*, представляющем собой код Ассемблера для машин серии ICL 1900.

1. *SETLABEL*

Элемент вариантного предложения, имеющий дело с этой командой, имеет вид

```
begin string label;
  read (label);
  print ((newline, label, X3, "NULL"))
end
```

Здесь значение *X3* — это строка, содержащая три пробела. Когда меткой станет *L23*, генерируется код

L23 NULL

где *NULL* — фиктивная команда (т. е. она ничего не делает). Для простоты допускается, что метки состоят из трех литер.

2. *GOTO*

Соответствующая часть транслятора выглядит следующим образом:

```
begin string label;
  read (label);
  print ((newline, X6, "BRN", X6, label))
end
```

Здесь значение *X6* есть строка, содержащая шесть пробелов. Если меткой будет *L23*, генерируется код

```
BRN L23
```

что означает «переход к метке *L23*».

3. *Вызов стандартного знака операции*

Параметрами являются конкретный знак операции, адреса операндов и адрес результата. Знак операции может использоваться при вызове элемента внутреннего вариантного предложения для генерации соответствующего кода. Например, если знак операции — *II+* (сложение двух целых чисел), код может генерироваться так:

```
begin address add1, add2, add3;
  read ((add1, add2, add3));
  convert (add1, 1);
  convert (add2, 2);
  convert (add3, 3);
  print ((newline, X6, "LDX  1 0(1)",
    newline, X6, "ADX  1 0(2)",
    newline, X6, "STO  1 0(3)"))
end
```

Допускается, что все адреса базированы в стеке. В результате выдается код

```
LDX 1 0(1)
ADX 1 0(2)
STO 1 0(3)
```

Первая команда кода *PLAN* загружает в регистр *1* значение адреса, на который указывает регистр *1*. Вторая команда складывает значение, содержащееся в адресе, на который указывает регистр *2*, со значением, содержащимся в регистре *1*. Третья команда помещает в память содержимое регистра *1* по адресу, на который указывает регистр *3*.

Convert также может генерировать какой-либо код. Например, если бы *add1* находился в стеке идентификаторов в блоке *1* и имел вид

```
(idstack, 1, 4)
```

то генерировался бы следующий код:

```
LDX 1 DISPLAY+1
ADN 1 4
```

В результате в регистр *1* сначала загружались бы значение *DISPLAY+1* и указатель на начало рамки стека, соответствующий

блоку *1*, а затем к нему добавилось бы смещение адреса, и в итоге регистр *1* указывал бы на *add1*. Если бы *add1* находился в рабочем стеке, в регистр пришлось бы также добавить размеры статических областей, находящихся под областью рабочего стека в рамке.

4. *JUMPF*

Параметрами служат метка и адрес времени прогона. Результатом команды является переход к метке, если значение, находящееся в адресе, окажется нулем («ложью»). Соответствующая часть транслятора имеет вид

```
begin string label;
  address add1;
  read ((label, add1));
  convert (add1, 1);
  print ((newline, X6, "LDX  1  0(1)", newline, X6, "BZE  1", X2, label))
end
```

При метке *L23*, кроме вида, полученного от вызова *convert*, выдается также код

```
LDX 1 0(1)
BZE 1 L23
```

(*BZE* есть мнемоническое обозначение перехода при нулевом значении).

5. *ASSIGN*

Параметрами являются целое число (представляющее тип или указатель на таблицу видов), исходный адрес *S* и адрес получателя *D*. Во время компиляции выполняется действие:

```
begin int m; address add1, add2;
  read ((m, add1, add2));
  convert (add1, 1);
  convert (add2, 2);
  scan (m)
end
```

Scan генерирует код для переписывания элемента типа *m* из адреса, на который указывает регистр *1*, по адресу, указываемому регистром *2*. Если тип простой, например **real** или **int**, то для этого потребуется лишь переписать значения одного или более последовательных адресов с помощью команды *MOVE* в коде *PLAN*. Например, *MOVE 1 2* перешлет содержимое двух последовательно расположенных адресов, начиная с адреса, на который указывает регистр *1*, в два последовательно расположенных адреса, начиная с адреса, на который указывает регистр *2*, причем конечные значения этих регистров не изменятся. Вообще тип может быть структурой или массивом. В случае структуры *scan* применяется рекурсивно к каждому из ее элементов, увеличивая содержимое регистров *1* и *2* после обращения с ними. Обработка массива, однако, сложнее. Очевидно, что статическую часть (т. е. описатель) переписывать необходимо, но нужно переписывать и динамическую часть (т. е. элементы самого массива). Для переписывания динамической части в регистры *1* и *2* вызывается *scan* с соответствующими указателями. Однако прежние значения регистров *1* и *2* не должны быть потеряны (представим, что массив был элементом какой-то структу-

ры), и для хранения их значений, кроме всего прочего, требуется стек. Он должен иметь произвольный размер, так как типы могут быть произвольной сложности. Тем не менее каждая реализация, по-видимому, накладывает свои (произвольные) ограничения на размер этого стека.

11.3. ОПТИМИЗАЦИЯ ОБЪЕКТНОГО КОДА

Как отмечалось в гл. 10, компиляторы отличаются по степени оптимизации выдаваемого объектного кода, направленной на то, чтобы программа проходила за возможно более короткое время. Выполнение некоторых видов локальной («узкой») оптимизации почти всегда оправдано. Опишем кратко такую оптимизацию. Рассмотрим три последовательности объектного кода:

1. *LDX 1 2*
   LDX 2 1

После загрузки содержимого регистра *2* в регистр *1*, несомненно, последует загрузка содержимого регистра *1* в регистр *2*, что будет указываться в следующей же команде. Вторая команда (тем более, что она не имеет метки) явно избыточна и ее можно исключить.

2. *BRN L23*
   команды без меток

Здесь *BRN* обозначает безусловный переход. Ряд команд без меток, которые следуют за безусловным переходом, выполнить нельзя и поэтому их можно исключить.

3. *BRN L2*
   L2 BRN L1

В этом случае тот же результат можно получить, заменив первую команду на *BRN L1*. Такие виды оптимизации выполняются во время генерации кода. Они легко осуществимы и недороги в смысле времени компиляции.

Упражнения

11.1. Каковы преимущества и недостатки метода компиляции в код Ассемблера, а не непосредственно в машинный код?

11.2. Напишите алгоритмы трансляции следующих команд промежуточного кода, которые были приведены в гл. 10:

1. *MOVE*
2. *JUMPG*

11.3. Какое дополнительное действие придется предпринять процедуре *convert* при обращении с косвенными адресами?

11.4. Какие осложнения может вызвать передача меток в виде целых чисел (т. е. *23*, а не *L23*) в командах промежуточного кода?

11.5. В различных алгоритмах трансляции команд промежуточного кода появляются описания одних и тех же идентификаторов. Как можно этого избежать? Каковы преимущества и недостатки этого подхода?

11.6. Как вы предполагаете, при каких обстоятельствах транслятор будет генерировать вызовы подпрограмм в коде Ассемблера?

11.7. В результате выполнения процедуры *convert* в регистр могло бы помещаться значение, а не указатель на это значение. Объясните, почему первый вариант не принят.

11.8. Присвоение в Алголе 68, например, обычно требует просмотра видов. В каких еще случаях может оказаться необходимым просмотр видов?

11.9. Укажите, что может потребоваться для реализации «узких» видов оптимизации, описанных в разд. 11.3?

11.10. Третья оптимизация в разд. 11.3 предполагает, что в двух последовательных командах происходят два безусловных перехода. Какое решение возможно в более общем случае?

Глава 12

ИСПРАВЛЕНИЕ И ДИАГНОСТИКА ОШИБОК

В этой главе мы покажем, правда, с некоторым запозданием, как компилятор должен обращаться с программами, в которых имеются ошибки программирования. Разработчик компилятора обязан с самого начала предусмотреть такую возможность, поскольку многие программы (если не подавляющее большинство), представляемые компилятору, содержат по меньшей мере одну ошибку. До сих пор мы старались не касаться вопросов, связанных с обнаружением и исправлением ошибок, но сейчас хотим подчеркнуть, как важно для компилятора правильно реагировать на ошибки, чтобы не снизилась его практическая ценность.

Конечно, мы рассмотрим здесь только те ошибки, которые могут быть обнаружены компилятором или привести к ошибке во время прогона. Мы не будем обсуждать формальные методы доказательства правильности программ, и ошибки программирования, которые не влияют на действенность программы (такие, как написание «—» вместо «+»), останутся необнаруженными во время компиляции или во время прогона.

Если программа, представленная компилятору, строго говоря, написана не на исходном языке, «недружелюбный» компилятор может просто проинформировать пользователя об этом, не указав, где могла произойти ошибка. Большинство пользователей не удовлетворит (и вполне оправданно) такой подход, поскольку они ожидают от компилятора

а) точного указания, где находится (первая) ошибка программирования;

б) продолжения компиляции (или, по крайней мере, анализа) программы после обнаружения первой ошибки с целью обнаружения остальных.

Как мы увидим позднее (разд. 12.4), требование (а) не всегда можно выполнить, поскольку результат ошибки программирования может сказаться не обязательно *в том месте, где эта ошибка была допущена*. Что касается требования (б), то это тоже непростая задача. Если ход выполнения программы стал недействительным, то неясно, какое действие должен предпринять компилятор, чтобы продолжить ее анализ. Существуют средства включения, исключения или замены символов в ходе выполнения программы; различные методы исправления ошибок обсуждаются в разд. 12.4.

12.1. ТИПЫ ОШИБОК

Ошибки могут возникать в программе по ряду причин:

1. Программист не совсем понимает тот язык, на котором он пишет, и использует неправильную конструкцию программы.

2. Программист недостаточно осторожен в применении конструкции языка и забывает описать идентификатор или согласовать открывающую скобку с закрывающей и т. д.

3. Программист неправильно пишет слово языка или какого-либо другого символа в программе.

Ошибки, обусловленные этими тремя факторами, по-разному обнаруживаются компилятором. Ошибки первого типа, очевидно, вылавливаются синтаксическим анализатором, и генерируется сообщение с указанием того символа, на котором поток программы стал недействительным. Ошибки второго типа распадаются на две категории. Те, которые относятся к контекстным средствам языка, например отсутствие описания идентификатора, обнаруживаются одной из процедур выборки таблицы символов, вызываемой анализатором во время синтаксического анализа. Такие ошибки, как недостающие скобки, могут быть обнаружены самим анализатором либо (в случае согласования скобок) во время выполнения специальной фазы или прохода, предназначенного для согласования скобок (см. разд. 7.1). Ошибки третьего типа обычно выявляются в процессе лексического анализа, хотя в случае неправильного написания идентификатора вновь потребуется выполнить одно из действий с таблицей символов.

Существуют ошибки еще одного типа, когда программа пытается выполнить деление на нуль, считывание за пределами файла или что-либо подобное. Они называются ошибками времени прогона, и обычно их нельзя обнаружить в процессе компиляции.

Далее в настоящей главе мы рассмотрим, как обнаруживаются различные ошибки — лексические, ошибки согласования, синтаксические, контекстные и ошибки времени прогона — и как о них сообщается программисту в понятной для него форме. Мы также обсудим методы исправления этих ошибок.

12.2. ЛЕКСИЧЕСКИЕ ОШИБКИ

Задача лексического анализатора — сгруппировать последовательности литер в символы исходного языка. При этом он работает исключительно с локальной информацией. В его распоряжении имеется очень небольшой объем памяти, и он не осуществляет никакого предварительного просмотра. Следовательно, те случаи, когда лексический анализатор окажется не в состоянии сгруппировать какие-либо последовательности литер в символы, будут свидетельствовать об ошибке. Лексические ошибки могут быть разными:

1. Одна из литер оказывается недействительной, т. е. она не может быть включена ни в один из символов. В таком случае лексический анализатор либо игнорирует эту литеру, либо заменяет ее какой-либо другой.

2. При попытке собрать слово языка, например выделенное слово в Алголе 60, выясняется, что последовательность букв не соответствует ни одному из этих слов. Тогда можно воспользоваться алгоритмом подбора слова, чтобы идентифицировать слово, имеющее несколько иное написание. Если алгоритм не сработает, то, допуская, что слова языка выделяются кавычками, как в *'BEGIN'*, стоит проверить, не начинается ли последовательность букв с какого-либо слова языка. Если это имеет место, то подстановка одной или двух кавычек сделает нашу

последовательность действительной. Например, в Алголе 60 у лексического анализатора возникнет затруднение с

а) *'BEGIN INTEGER'*

или

б) *'REALAB*

Однако подстановкой двух кавычек в (а) и одной кавычки в (б):

а) *'BEGIN' 'INTEGER'*

и

б) *'REAL' AB*

мы решаем проблему.

В Алголе 68 существует бесконечное число возможных выделенных слов, и «неправильно написанное» выделенное слово будет принято лексическим анализатором.

3. Собирая числа, лексический анализатор может испытывать затруднения в обращении с такой последовательностью, как *42.34.41*. Возможное решение здесь — допустить, какая бы ошибка ни была, что предполагалось одно число, и предупредить программиста, что вместо этого числа принято конкретное значение по умолчанию.

4. Отсутствие в программе какой-либо литеры часто приводит к тому, что лексический анализатор не может отделить один символ от другого. В некоторых случаях лексический анализатор принимает два символа за один. Например, если в $A+B$ опущен знак «+», то лексический анализатор просто пропустит идентификатор *AB*, не оповещая об ошибке на этой стадии. Однако отсутствие знака «+» в $1+A$ вызовет ошибку, хотя лексический анализатор не будет знать, к какой группе он должен отнести *1A* — к недопустимым идентификаторам, недопустимым числам или еще к чему-либо. Иногда лучше разбить недопустимые последовательности на более короткие допустимые. Так, *1A* станет числом *1*, за которым следует идентификатор *A*. Тогда синтаксический анализатор обнаружит ошибку.

5. Обычно проблему для лексического анализатора создают недостающие кавычки строки, как, например, в

string *food* := *"BREAD*

Следовательно, в остальной части программы открывающие и закрывающие кавычки могут быть перепутаны, т. е. то, что заключено в кавычки, окажется вне кавычек и наоборот. В результате почти наверняка на нас лавиной обрушатся сообщения об ошибках. «Смышленый» анализатор смог бы обнаружить неправдоподобную последовательность литер внутри кавычек (например, **end**) и исправить ошибку, поставив в нужном месте кавычки. Подобная проблема возникает в связи с комментариями в Алголе 60 и Алголе 68 и в связи с форматами в Алголе 68. Эти проблемы могли бы быть не такими серьезными, если бы разработчики языков использовали *разные* символы для обозначения начала и конца комментариев.

Существуют компиляторы, которые завершают свою работу тем проходом, где обнаружена ошибка. Однако если задаться целью обнаружить максимальное число ошибок, то желательно, чтобы компилятор продвинулся в своей работе как можно дальше, если это вообще возможно. Поэтому лексический анализатор должен передать следующему проходу (или фазе) последовательность действительных символов (а необязательно действительную последовательность символов). Для правильных в лексическом смысле программ это не пред-

ставляет трудностей. При лексически неправильных программах придется или игнорировать последовательность литер, или включать их, может понадобиться изменить написание символов, а строки разбить на действительные символы. Игнорирование последовательностей литер, пожалуй, самое простое средство, но почти наверняка оно приведет позднее к возникновению синтаксических ошибок. Методы исправления лексическим анализатором недопустимых входов зависят от обстоятельств, и на практике их выбор почти всегда определяется компилируемым языком и тем, как программисты предполагают им воспользоваться. Имеет смысл разработать такой вариант, чтобы лексический анализатор, когда он не в состоянии распознать какой-либо действительный символ или символы из строки литер, выдавал специальный символ «не знаю», который интерпретировался бы синтаксическим анализатором, имеющим обычно список действительных символов продолжения, по своему разумению.

12.3. ОШИБКИ В УПОТРЕБЛЕНИИ СКОБОК

Ошибки, связанные с употреблением скобок, обнаружить относительно легко. Некоторые компиляторы имеют фазу, предшествующую полному синтаксическому анализу, на которой проводится согласование скобок (или их идентификация). Если применяются скобки только одного типа, например «(» и «)», то проверку можно осуществить с помощью целочисленного счетчика. Этот счетчик первоначально устанавливается на нуль, затем увеличивается на единицу для каждой открывающей скобки и уменьшается на единицу для каждой закрывающей скобки. Последовательность скобок считается допустимой в том случае, когда

1) счетчик ни при каких обстоятельствах не становится отрицательным;

2) при завершении работы счетчик будет на нуле.

Допустимы следующие структуры скобок:

```
( (     )  (     )  (       ))
(   }      ((((  ))))
```

Приводимые ниже структуры

```
(     ) (   )) ((
(     )    ((    )
```

недопустимы, так как счетчик в первом случае оказывается отрицательным, а во втором не является нулевым при завершении работы.

В большинстве языков программирования встречаются различные типы скобок, например

```
|          |
(          )
if         fi
case       esac
begin      end
```

Применения описанного выше алгоритма, рассматривающего все открывающие скобки как «(», а все закрывающие как «)», недостаточно, поскольку последовательность

```
[(        ])
```

не следует считать допустимой. Необходимо согласовывать каждую закрывающую скобку с соответствующей открывающей. Этот алгоритм пользуется стеком и читает скобочную структуру слева направо, помещая каждую открывающую скобку в вершину стека. Когда встречается закрывающая скобка, соответствующая открывающая скобка удаляется из стека. Последовательность скобок является допустимой, если

1) при чтении закрывающей скобки не окажется, что она не соответствует открывающей скобке, помещенной в вершине стека, или если стек не будет пустым;

2) при завершении работы стек станет пустым.

Ошибка в употреблении скобок должна отразиться в четком сообщении компилятора, таком, как

BRACKET MISMATCH

Если ошибка возникла из-за того, что не достает закрывающей скобки, то тип недостающей скобки можно вывести на основании той скобки, которая находится в вершине стека. Один из возможных путей исправления ошибки заключается в том, что берется предполагаемая недостающая закрывающая скобка, открывающая скобка удаляется из стека, и выдается сообщение с указанием предполагаемого источника ошибки, например (в предположении, что в верхней части стека находилось *IF*)

MISSING FI?

Диагностическое сообщение, однако, появится не в том месте, где был допущен пропуск скобки, так как ошибка останется незамеченной до тех пор, пока не встретится другая закрывающая скобка (иного типа). При продолжении синтаксического анализа желательно, чтобы скобочная структура была исправлена. Рассмотрим (недопустимый) фрагмент программы:

if b **then** x **else** $(p+q\times r\uparrow 2$ **fi**

Здесь пропущена закрывающая скобка «)». Это не обнаружится до тех пор, пока не встретится **fi**. Однако не ясно, где должна стоять эта скобка: после r, после q или, что менее вероятно, после p или 2. Выяснить, что предполагал программист, невозможно. Поэтому самый легкий способ «исправления» — поставить закрывающую скобку непосредственно перед **fi**. По крайней мере, это позволит синтаксическому анализатору продолжать работу без выдачи другого сообщения об ошибке из-за недостающей скобки. Существуют более сложные алгоритмы исправления структур [4].

12.4. СИНТАКСИЧЕСКИЕ ОШИБКИ

Термин «синтаксическая ошибка» употребляется для обозначения ошибки, обнаруживаемой контекстно-свободным синтаксическим анализатором. LL(1)- и LR(1)-анализаторы обладают этим важным свойством — обнаруживать синтаксически неправильную программу на первом недопустимом символе, т. е. они могут генерировать сообщение при чтении символа, который не должен следовать за прочитанной к тому времени последовательностью символов. Некоторые же анализаторы (например, анализаторы с приоритетами знаков операций)

такого свойства не имеют. Далее в этом разделе мы полагаем, что синтаксические ошибки обнаруживаются при первой возможности.

При описании методов исправления ошибок мы будем ориентироваться на LL(1)-анализаторы, хотя большинство положений легко распространить и на LR(1)-анализаторы. Что касается ошибок в употреблении скобок, то синтаксические ошибки не обязательно обнаруживаются вблизи от того места, где программист допустил ошибку. Ошибка в виде пропуска или неправильного употребления, допущенная на более раннем этапе, может проявиться совсем в другом месте программы. Проиллюстрируем это положение с помощью Алгола 68. Рассмотрим цикл

while $x>y$ **do** *something* **od**

Синтаксис этого языка не разрешает ставить перед **do** знак «;». Однако мы покажем, что если «;» стоит, анализатор не сможет обнаружить ошибку в *этом месте.*

В Алголе 68 можно записать бесконечный цикл (т. е. цикл, который заставит программу работать бесконечное число раз, если ее не остановит такая конструкция, как **goto**) следующим образом:

do
something else
od

В программе могут оказаться два вложенных цикла:

while $x := 1$ **do** *something else* **od** ; $x>y$
do *something* **od**

где частью **while** внешнего цикла является последовательное предложение (точнее, выясняющее предложение), которое содержит элемент, представляющий собой бесконечный цикл. Интересно еще и то, что этот фрагмент программы включает последовательность

:do

Поэтому, если бы мы написали

while $x>y$ **; do** *something* **od**

то никакого сообщения об ошибке при встрече «;» или **do** не последовало бы, так как на данной стадии синтаксический анализатор не может исключить вероятность того, что **do** служит началом бесконечного цикла, и не связано с предшествующим **while.** Конечно, следует ожидать появления другого **do** в пару к **while,** и это должно произойти прежде, чем закроется скобочная структура, открытая перед **while.** Ошибка (если она имела место) обнаружится при чтении какой-либо закрывающей скобки и о ней будет выдано сообщение, что может случиться через много строк после того места, где была поставлена недопустимая точка с запятой.

Сообщив о синтаксической ошибке, анализатор в большинстве случаев постарается продолжить разбор. Для этого ему может понадобиться исключить какие-либо символы, включить какие-либо символы или изменить их (исключить и включить). Существует ряд стратегий исправления ошибок. Некоторые простые стратегии описываются здесь. Практически все они хорошо срабатывают в одних случаях и плохо — в других. «Хорошая» стратегия заключается в том, чтобы обнаружить как можно больше синтаксических ошибок и генерировать как можно меньше сообщений в связи с каждой синтаксической ошибкой. Обычно

наилучшими являются методы, зависимые от языка, т. е. от знания исходного языка и от того, как он употребляется.

1. *Режим переполоха*

Один из наиболее общеупотребительных методов исправления синтаксических ошибок носит название *режима переполоха*. При появлении недопустимого символа весь последующий исходный текст, вплоть до соответствующего ограничителя, например «;» или **end**, игнорируется. Ограничитель заканчивает какую-то конструкцию языка, и элементы удаляются из стека разбора до тех пор, пока не встретится адрес возврата. Этот элемент тоже удаляется из стека, а разбор продолжается, начиная с адреса в таблице разбора, содержащего следующий входной символ. Такой метод довольно легко реализуется, но имеет серьезный недостаток: длинные последовательности кода, соответствующие игнорируемым символам, не анализируются.

2. *Исключение символов*

Этот метод также легко реализуется и не требует изменения стека разбора. Когда считывается недопустимый символ, и он сам, и все последующие символы исключаются из исходной строки до тех пор, пока не встретится допустимый. Хотя при таком методе могут исключаться длинные последовательности символов, в определенных случаях он весьма эффективен. Например, в

$$c := d+3\ ;\ \textbf{end}$$

где «;» является недопустимой, исправление ошибки — идеальное. Однако исключение скобок обычно нарушает скобочную структуру программы и приводит к дальнейшим синтаксическим ошибкам.

3. *Включение символов*

LL(1)-анализатор на любой стадии разбора должен иметь наготове множество действительных символов продолжения. В некоторых случаях оправдано исправление программы путем подстановки одного из этих символов перед недопустимым символом, который вызвал ошибку. Например, последовательность

end begin

никогда не будет допустимой в Алголе 68. Однако включение «;» между **end** и **begin** позволит анализатору продолжить работу. (Последующая информация, имеющаяся в распоряжении анализатора, может подсказать, что здесь более уместна запятая.) В целом же мы можем утверждать (в отношении Алгола 68), что если встречается синтаксическая ошибка и недопустимым считается один из символов

begin, if, case, (

а предыдущим считанным — один из символов

end, fi, esac,)

то при включении «;» (или «,») анализатор продолжит работу. Конечно, могла иметь место и неправильная подстановка, даже если ана-

лизатор продолжил разбор. Например, программист мог пропустить между символами **end** и **begin** символ «+» или «—».

4. *Правила для ошибок*

Один из способов исправления некоторых типов синтаксических ошибок заключается в расширении синтаксиса языка за счет включения в него программ, содержащих данные ошибки. Это не значит, что ошибки пройдут незамеченными, так как в грамматику могут быть включены сообщающие о них действия, но анализатор не будет считать такой вход недопустимым, и не потребуется никаких исправлений. Так можно обращаться, например, с ошибками типа «;» перед **end**, как в Алголе 68, или пропуск «;» перед последовательностью

L : **end**

в Алголе 60. Дополнительные правила, включенные в грамматику, обычно называются «правилами для ошибок». Они неизбежно приводят к увеличению грамматики (и анализатора), и поэтому включать их следует только для наиболее часто встречающихся ошибок программирования. Необходимо также следить за тем, чтобы при включении этих дополнительных правил грамматика не стала неоднозначной.

Предупреждения

Наряду с сообщением о синтаксических ошибках анализатор может выдавать предупреждения, когда ему встречается допустимая, но маловероятная последовательность символов, например

;**do**

Эти предупреждения, как мы видели, могут быть в программах, написанных на Алголе 68, но в синтаксически правильных программах используются редко. В грамматику может вводиться действие, которое при каждом считывании **do** проверяет предыдущий символ, не является ли он «;».

Сообщения о синтаксических ошибках

Всякий раз при обнаружении анализатором синтаксической ошибки должно печататься соответствующее сообщение. Например, может выдаваться такое сообщение, как

SYNTAX ERROR IN LINE 22

или местоположение ошибки может описываться полнее:

LINE 22 SYMBOL 4

В любом из этих случаев пользователь может быть недоволен тем, что сообщение не вполне ясное, так как не указывает, в чем заключается ошибка программиста. Как уже отмечалось, фактическая ошибка программирования могла произойти гораздо раньше, анализатор же сообщает об ошибке только тогда, когда ему встречается недопустимый символ. Если программист представляет анализатору программу, написанную, как предполагается, на ПЛ/1, но имеющую синтаксическую ошибку, компилятор не сможет решить, какую программу на ПЛ/1 программист должен был бы написать. Единственное, что компилятор

смог бы сделать, это принять решение о «ремонте» на минимальном расстоянии, т. е. о ремонте, требующем минимальное число включений символов в текст программы и исключений из него, дающих синтаксически правильную программу. Мы употребляем здесь термин «ремонт на минимальном расстоянии», а не (как некоторые авторы) «исправление на минимальном расстоянии», поскольку нет никакой гарантии, что этот ремонт действительно позволит получить ту программу, которая имелась в виду в самом начале. Цель ремонта — обеспечить анализатору условия для продолжения анализа программы. Алгоритмы ремонта на минимальном расстоянии приводятся в [4].

Хотя теоретически ремонт на минимальном расстоянии кажется привлекательным, его реализация неэффективна, так как приходится часто возвращаться назад по уже проанализированным частям программы и отменять выполненные ранее компилятором действия. Большинство компиляторов не берется за такой ремонт. Единственное исправление, которое они осуществляют, — это вставка, исключение или изменение символов *в том месте, где обнаружена ошибка*. В этом случае компилятор не может предоставить пользователю иной информации, кроме точного указания о том, где обнаружена ошибка. Компилятору может быть известен еще и контекст, в котором обнаружена ошибка; например, она могла произойти в пределах присваивания, в границах массива или в вызове процедуры. Такая информация не всегда оказывается ценной для пользователя, но она показывает, какой тип конструкции пытался распознать анализатор, когда обнаружил ошибку, а это поможет найти фактическую ошибку программирования. Можно было бы также сообщить пользователю, какие символы допустимы при встрече недопустимого символа, что тоже помогло бы пользователю в исправлении программы. Если анализатор способен (в некоторых случаях) сделать разумное предположение о том, какая фактическая ошибка программирования была допущена, он может исправить программу (для последующих проходов), но это возможно лишь тогда, когда он четко сообщает пользователю, какое действие при этом выполняется.

Для исправления программы (но не ремонта) необходимо знать истинные намерения программиста. Так, на экзамене по программированию студенту обычно предлагается классическая задача: «Исправьте следующую программу...». Однако что ожидается от студента? Должен ли он найти исправление на минимальном расстоянии (оно ведь может быть не единственным), требуемое для получения синтаксически правильной программы, или ему нужно изменить текст, чтобы получить программу, которую, по его мнению, намеревался написать программист (или экзаменатор)? Формулировка задачи представляется неточной.

12.5. КОНТЕКСТНО-ЗАВИСИМЫЕ ОШИБКИ

Мы уже видели, что некоторые конструкции типичных языков программирования нельзя описать с помощью контекстно-свободной грамматики, поэтому анализатор, как правило, базируется на контекстно-свободной грамматике, генерирующей сверхмножество компилируемого языка. Следовательно, с точки зрения таблицы разбора программы с неописанными идентификаторами и т. п. синтаксически правильны. Однако любые такие контекстно-зависимые ошибки будут обнаружены действиями, включаемыми в контекстно-свободную грамматику и вы-

зываемыми анализатором, который запрашивает таблицу символов и т. п. Об ошибках подобного типа обычно выдаются четкие сообщения при наличии таблицы идентификаторов, созданной во время лексического анализа, например

IDENTIFIER xyz NOT DECLARED
TYPES NOT COMPATIBLE IN ASSIGNMENT

а на Алголе 68:

ILLEGAL MODE DECLARATION
BALANCING NOT POSSIBLE

Так как сам анализатор ошибку не обнаружил, никакого исправления не требуется. Однако если не принять соответствующие меры, то одна ошибка может повлечь за собой лавину сообщений об ошибках. Например, если в описании идентификатора допущена ошибка в написании

int *wednesday*

каждая прикладная реализация *wednesday* может вызвать сообщение об ошибке

IDENTIFIER wednesday NOT DECLARED

Во избежание этого при первом же появлении неописанного идентификатора с указанным именем идентификатор *wednesday* должен включаться в таблицу символов. В таблицу также должен помещаться тип, соответствующий неописанному идентификатору. Компилятор располагает недостаточной информацией, чтобы заключить, какой тип идентификатора предполагается, поэтому многие компиляторы принимают стандартный тип **int** или **real.** Лучше всего иметь для этого специальный тип, скажем, **sptype**, который будет ассоциироваться с такими идентификаторами; **sptype** обладает следующими свойствами:

1. Его можно приводить к любому типу/виду.
2. Если значение типа **sptype** оказывается операндом, знак операции идентифицируется с помощью другого операнда (при его наличии), причем любая неоднозначность разрешается произвольно.
3. Применительно к анализатору значение типа **sptype** выбирается или вырезается и т. д., хотя, конечно, выдача соответствующего кода может быть невозможной.

Не исключена и другая ошибка: в одном и том же блоке идентификатор описывается дважды. Всякий раз при описании идентификатора желательно проверять, не был ли идентификатор с этим же именем уже описан в текущем блоке. Если он был описан ранее, то следует генерировать такое сообщение:

IDENTIFIER blank ALREADY DECLARED IN BLOCK

Чтобы избежать неоднозначности, в таблице символов на каждом уровне блоков для каждого идентификатора должен появляться один элемент. Что предпринимает компилятор, когда он встречает второе или последующие описания идентификатора в блоке? Для однопроходного компилятора лучше сохранить существующий элемент, так как им уже мог воспользоваться анализатор, а замена этого элемента информацией, полученной из более позднего описания, может создать впечатление, что компилятор действует нелогично. Оптимальным вариантом было бы проведение во время компиляции подробного анали-

за части программы, что позволило бы посмотреть, как этот идентификатор используется в блоке, и решить, какое из описаний ему более всего соответствует.

Ошибки, связанные с употреблением типов

Правила, определяющие, где в программе возможно появление значений различных типов, в большинстве случаев не являются контекстно-независимыми. Если идентификатор описан таким образом, что ему могут присваиваться значения в виде целых чисел, то во многих языках попытка присвоить ему литерное значение будет считаться недопустимой. Для языков с явно выраженными типами (например, Паскаль, Алгол 68) такая ошибка обнаруживается во время компиляции с помощью таблицы символов. В Алголе 68 преобразование значения одного вида в другой осуществляется с помощью последовательности приведений (см. разд. 10.4). В тех случаях, когда в программе априорный вид значения не может быть приведен в соответствующий апостериорный вид, компилятор Алгола 68 выдает следующее сообщение:

MODE char CANNOT BE COERCED TO int

Это способствует идентификации ошибки, но вряд ли может помочь начинающему программисту, который не знаком с таким жаргоном. Предположим, была допущена ошибка:

$i := c$

где i есть вид **ref int**, а c — вид **char**. Тогда более понятным становится сообщение

MODES NOT COMPATIBLE IN ASSIGNMENT

Как мы уже видели (в разд. 8.2), описания видов в Алголе 68 определяются частично контекстно-свободными правилами и частично контекстно-зависимыми. Например, следующие описания видов допустимы с точки зрения контекстно-свободных правил, но требованиям контекстно-зависимых правил не удовлетворяют:

```
mode x=y
mode y=ref x
```

Поскольку в описание вида может быть вовлечен другой вид, определяемый пользователем (взаимно рекурсивные виды), такие ошибки вообще нельзя обнаружить до тех пор, пока все виды не будут полностью описаны. Поэтому сообщения об этих ошибках могут выдаваться между проходами компилятора.

Существуют и другие ошибки, связанные с контекстно-зависимыми аспектами типичных языков:

1) неправильное число индексов массива;

2) неправильное число параметров для вызова процедуры или функции;

3) несовместимость типа (или вида) фактического параметра в вызове с типом формального параметра;

4) невозможность определения знака операции по его операндам.

Обычно, когда встречаются такие ошибки, компилятор может выдавать четкие сообщения.

12.6. ОШИБКИ, ДОПУСКАЕМЫЕ ВО ВРЕМЯ ПРОГОНА

Во время прогона в программах могут возникать ошибки, которые нельзя предусмотреть в процессе компиляции. Конечно, при компиляции можно обнаружить явное деление на нуль, например $m/0$. Но выражение m/n также может повлечь за собой деление на нуль (если n окажется нулем, когда будет вычислено выражение) и это нельзя (как правило) обнаружить во время компиляции. При прогоне возможны другие ошибки:

1) нахождение индекса массива вне области действия;

2) целочисленное переполнение (вызванное, например, попыткой сложить два наибольших целых числа, допускаемых реализацией);

3) попытка чтения за пределами файла.

В языках с динамическими типами до времени прогона нельзя обнаружить более широкий класс ошибок. Сюда входят ошибки в употреблении типов, связанные с присваиваниями, и идентификация знаков операций.

Разработчики языков обычно стараются предотвратить возможность возникновения ошибок, которые нельзя обнаружить до прогона [54]. Одно из решений — постараться дать исчерпывающую формулировку задачи, например результат деления чего-либо на нуль определить как нуль, выходящий за пределы области действия, индекс считать эквивалентным какому-нибудь значению в пределах области действия, при попытке чтения за пределами файла выполнять некоторое стандартное действие и т. д. В Коболе программист специфицирует действие, которое следует выполнить при достижении конца файла, и аналогичным образом он мог бы определять необходимые действия в случае других ошибок, встречающихся во время прогона.

Однако такая исчерпывающая формулировка задачи имеет свои «подводные камни»: могут остаться незамеченными ошибки программирования или ошибки в данных. Программисты обычно не ожидают, что во время прогона их программ произойдет деление на нуль, — им об этом нужно сообщать. Тем не менее весьма нежелательно, чтобы из-за этого прерывалось выполнение программы. Компромиссное решение — напечатать сообщение об ошибке во время прогона, когда она возникает, но позволить программе выполнить какое-либо стандартное действие, чтобы она могла продолжать работу и находить дальнейшие ошибки. Такой подход аналогичен исправлению ошибок на фазе разбора при компиляции.

В случае ошибки, возникающей в процессе прогона, не всегда можно четко объяснить программисту, что именно сделано неправильно. К этому моменту программа уже транслирована в машинный код, а программисту, вероятно, понятны только ссылки на версию в исходном коде. Поэтому система, работающая при прогоне, должна иметь доступ к таблице идентификаторов и другим таблицам и следить за номерами строк в исходной программе. Таблицы, требуемые для диагностики, к началу прогона программы могут уже не находиться в основной памяти, но в случае ошибки будут туда загружаться. Кроме сообщения об ошибке, можно также выдать на печать профиль программы на то время, когда произошла ошибка, включая значения всех переменных. Это помогает идентифицировать ошибки, вызванные неинициализированными переменными.

12.7. ОШИБКИ, СВЯЗАННЫЕ С НАРУШЕНИЕМ ОГРАНИЧЕНИЙ

До сих пор мы считали, что компилятор должен быть в состоянии скомпилировать *любую* программу, написанную на соответствующем исходном языке. Однако это не всегда так, хотя бы из-за конечного размера машины, на которой он работает. Хороший компилятор имеет мало произвольных ограничений, но если ограничения вводятся, они должны быть такими, чтобы устраивать подавляющее большинство программ среднего размера. Возможны ограничения на

1) размер программ, которые можно скомпилировать;
2) число элементов в таблице символов или идентификаторов;
3) размер стека разбора или других стеков времени компиляции.

Если один и тот же объем памяти отводится под совместное пользование для различных таблиц, то может быть ограничен общий объем, а не объем, занимаемый конкретной таблицей.

Существует вероятность того, что программа заставит нарушить какое-нибудь из ограничений. В этом случае важно, чтобы компилятор выдавал четкое сообщение пользователю, какое именно ограничение нарушено.

Упражнения

12.1. Считаете ли вы, что реализация хорошего метода исправления ошибок замедлит компиляцию программы, не имеющей ошибок?

12.2. Некоторые языки используют «зарезервированные» идентификаторы в качестве слов. Какие преимущества это дает в смысле обнаружения ошибок?

12.3. Какие средства языка могут спровоцировать программиста на ошибку или не позволят ему допустить ее?

12.4. Какие средства языка могут способствовать полезным сообщениям об ошибках, а какие нет?

12.5. Объясните, как при наличии фиктивного оператора (не состоящего из каких-либо символов) в Алголе 60 и Паскале может остаться незамеченным неправильное употребление точек с запятой.

12.6. Какое преимущество в отношении исправления ошибок предлагают ограничители комментария в Паскале («{», «}») по сравнению с ограничителями в Алголе 68 (**co, co**)?

12.7. Считаете ли вы, что стратегия исправления ошибок не должна зависеть от исходного языка?

12.8. Возможные источники ошибок, возникающих во время прогона, можно обнаружить в процессе компиляции. Считаете ли вы это полезным?

12.9. В некоторых компиляторах можно отключить проверку индексов во время прогона. Как вы думаете, почему это делается и насколько это целесообразно, по вашему мнению?

12.10. Предложите, как можно написать компилятор с одним ограничением в отношении объема памяти, т. е. чтобы программы не могли бы компилироваться только из-за недостатка в объеме памяти, если бы *вся* доступная компилятору память была заполнена.

Глава 13

СОЗДАНИЕ НАДЕЖНЫХ КОМПИЛЯТОРОВ

Первоначально эта глава имела название «Создание правильных компиляторов». Однако такое название выглядело несколько претенциозно. Сказать с уверенностью, что какой-либо компилятор является правильным в том смысле, что любой вход в него даст соответствующий выход (объективный код или сообщения об ошибках), представляется невозможным в свете настоящего положения дел в данной области. Большинство компиляторов несовершенно в такой степени, что по меньшей мере несколько программ окажутся скомпилированными неправильно, не будут приняты компилятором, поставят компилятор в тупик или заставят его зациклиться. В гл. 1 мы обсуждали возможные цели проектирования компилятора, одной из которых является надежность. Здесь мы вновь вернемся к этому аспекту проектирования и обсудим взаимосвязь между формальным определением языка и его реализацией, модульным проектированием и верификацией компилятора.

13.1. ИСПОЛЬЗОВАНИЕ ФОРМАЛЬНОГО ОПРЕДЕЛЕНИЯ

В формальном определении языка описываются 1) его синтаксис и 2) его семантика. Что касается синтаксиса, то с точки зрения разработчика контекстно-свободные и контекстно-зависимые аспекты удобно рассматривать отдельно. Как мы уже видели, контекстно-свободную грамматику, соответствующую контекстно-свободным аспектам языка, обычно с помощью соответствующего генератора анализаторов можно преобразовать в контекстно-свободный синтаксический анализатор. Более того, привлечение к этому делу программы (а не вручную) дает еще большую уверенность в надежности получаемого анализатора. Двухуровневая грамматика, на которой базируется Алгол 68, представляет собой, конечно, обобщение контекстно-свободной грамматики, и поэтому нужно тщательно определить лежащую в ее основе контекстно-свободную грамматику, вводимую в генератор анализаторов. В значительной степени этого можно добиться, подавляя метапонятия и предикаты, связанные с видами, определяющими и прикладными реализациями, и т. д. Если допустить, что в генератор анализаторов подается «правильная» контекстно-свободная грамматика, можно надеяться, что полученный в результате анализатор будет достаточно надежным.

Конечно, в компиляторе должен быть предусмотрен разпознаватель полного языка (включая его контекстно-зависимые аспекты). Идея положить в основу такого распознавателя грамматику Хомского типа 0 или двухуровневую W-грамматику неосуществима. В этом случае распознаватель был бы эквивалентен машине Тьюринга, а из-за соображений эффективности желательно найти какую-то менее общую модель.

Например, большинство контекстно-зависимых аспектов Алгола 68 можно выразить с помощью атрибутивной грамматики [51]. Уже разработаны методы автоматического построения анализаторов на основании определенных классов атрибутивных грамматик [57].

Пересмотренное сообщение об Алголе 68 можно также реализовать непосредственным путем (а не через атрибутивные грамматики). Метапонятия, там, где они встречаются, обычно заменяются в контекстно-свободной грамматике действиями. Например, в

REF to MODE NEST assignation : REF TO MODE NEST destination,
становится *token, MODE NEST source.*

Действие после *destination* можно использовать для помещения в стек его вида, а действие после *source* — для проверки совместимости видов источника и получателя и для вывода последовательности приведений, применяемых к источнику и получателю. Таким образом, двухуровневую грамматику нетрудно в большей части преобразовать в контекстно-свободную с включенными в нее действиями, на основании которых строится анализатор, содержащий вызовы соответствующих действий.

Для усиления некоторых контекстно-зависимых условий в пересмотренном сообщении также используются предикаты. Например, в

MODE1 NEST source : strong MODE2 NEST unit,
where MODE1 deflexes to MODE2.

у нас имеется предикат

where MODE1 deflexes to MODE2

который позволяет обеспечить соотношения двух видов путем «приведения». При выполнении приведения можно воспользоваться действием, включенным в грамматику в соответствующем месте. Это — вызов процедуры *deflexes* с параметром *MODE1*. Определение предиката *deflexes* дано в сообщении ([55] разд. 4.7.1), а вызов соответствующей процедуры может быть рекурсивным.

Как уже отмечалось ранее (см. разд. 7.1), нельзя анализировать каждую программу на Алголе 68 за один проход по исходному тексту, так как в нужный момент может отсутствовать информация о видах значений и т. п. Проблема анализа двухуровневой грамматики описанным выше способом заключается в том, что выполнение некоторых действий компилятора по усилению контекстно-зависимых требований окажется невозможным, если в предыдущих проходах не будут построены таблицы символов и т. п. Эта проблема разрешима в тех случаях, когда на стадии анализа для каждого прохода компилятора используется одна и та же контекстно-свободная грамматика, но в каждом проходе включаются только те действия, которые имеют к нему отношение. Например, в один проход включаются действия по построению таблиц символов, в другой — действия по обращению к ним и т. д. Различные контекстно-свободные грамматики с включенными действиями, объединенные вместе, соответствуют двухуровневой грамматике из пересмотренного сообщения.

Такой подход требует от составителя компилятора достаточно глубокого понимания языка. Он должен суметь оценить задачи, которые придется выполнять компилятору в каждом проходе. Тем не менее анализатор можно создать и автоматически. В частности, процедуры, соответствующие предикатам, во многих случаях допустимо на-

писать непосредственно по сообщению (хотя иногда это приводит к значительному снижению эффективности). В гл. 10 было показано, что обнаружение последовательностей приведения тоже может основываться почти непосредственно на сообщении. Костер [40] описывает компилятор компиляторов, базирующийся на двухуровневых грамматиках, который использовался при создании Манчестерского компилятора Алгола 68 [5].

Конечно, разработчику компиляторов хотелось бы, чтобы формальное определение языка позволяло реализовать его более или менее непосредственным путем, хотя это в некоторой степени может противоречить другим требованиям, предъявляемым к формальному определению. Взаимосвязь между формальным определением языка и его реализацией рассматривается в [23], а способы выражения контекстно-зависимых аспектов Алгола 60 и Бейсика с помощью формальной нотации — в [58].

13.2. МОДУЛЬНОЕ ПРОЕКТИРОВАНИЕ

В предыдущих главах мы познакомились с различными процессами, которые осуществляет компилятор. В однопроходном компиляторе все действия выполняются параллельно, и даже в многопроходном компиляторе несколько действий может осуществляться параллельно в одном и том же проходе. Например, в одном проходе могут объединяться лексический и синтаксический анализы, а в другом — распределение памяти и генерация кода.

С целью обеспечения надежности, простоты обновления и т. п. желательно, чтобы коды для различных фаз компиляции, входящих в один проход, хранились по возможности отдельно. Часто этого можно добиться, если одна фаза будет вызывать другую как процедуру. Общеизвестно, например, что синтаксический анализатор вызывает лексический анализатор как процедуру. Можно привести и другой пример: когда две фазы обладают одинаковым статусом, используются сопрограммы или фазы считаются кооперированными последовательными процессами [18].

Если код какого-либо прохода включает в основном действия, вызываемые синтаксическим анализатором, а сами действия являются элементами вариантного предложения **case**, то для разделения кодов, соответствующих, скажем, двум фазам, составляющим этот проход, можно применить следующую стратегию. Предложение **case**, содержащее действия, вызываемые анализатором, имеет форму:

```
case i in
  action1,
  action2,
  action3,
    .
    .
    .
esac
```

Здесь *action 1* может быть таким:

```
begin prologue;
    phase1 (1);
    phase2 (1);
    epilogue
end
```

где *phase1* — процедура вида

```
proc phase1 = (int no) void:
begin case no in
        .
        .
        .
      esac
end
```

Так что вызов *phase1 (1)* повлечет за собой выполнение первого элемента предложения **case** в пределах *phase 1*. Процедура *phase 2* по форме аналогична *phase1*, а *prologue* включает такие действия, как удаление из стека значений времени компиляции, требуемых и для *phase1*, и для *phase2*. *Prologue* и *epilogue* введены с тем, чтобы эти фазы не выполняли одни и те же поддействия.

Как уже отмечалось, коды для различных фаз должны храниться по возможности раздельно. Более того, весьма желательно, чтобы эти фазы знали друг о друге как можно меньше. Например, синтаксическому анализатору вовсе не нужно знать, как лексический анализатор собирает литеры в символы; все что ему требуется, — это вызвать подпрограмму, которая выдаст следующий символ. Генератору кода не должно быть известно, как распределяются адреса, если он может получить их при необходимости из таблицы символов, а распределение адресов и помещение их там, где это нужно, в таблицу символов является задачей распределителя памяти. Ни генератору кода, ни распределителю памяти не требуется знать структуру таблицы символов, если имеются соответствующие подпрограммы обращения к таблице и ее обновления.

В таких языках, как Ада, состоящих из модулей или пакетов, можно прятать идентификаторы от тех частей программы, для которых доступ к ним не нужен. Идентификаторы можно прятать также в Алголе 68R. Когда какой-то сегмент программы компилируется отдельно, идентификаторы будут видны следующим сегментам, только если они явно включены в список **keep,** ассоциируемый с определенным сегментом.

Следующий сегмент объявляет, что массив должен использоваться в виде стека вместе с указателем стека и процедурами *push* и *pop* для помещения элемента в стек и удаления его оттуда:

```
stackseg
begin [1:50] int stack;
  int stackptr := 0;
  proc push = (int v) void:
  begin if stackptr ≠ 50
    then stackptr plusab 1;
    stack [stackptr] := v
    else full
    fi
  end;
  proc pop = int;
  begin int v;
    if stackptr ≠ 0
    then v := stack [stackptr];
    stackptr minusab 1
    else empty
    fi;
    v
  end;
  skip
end
keep push, pop
```

Здесь, как обычно, *full* и *empty* — процедуры обращения с переполнением и незаполнением соответственно.

После того как сегмент скомпилирован и помещен в так называемый альбом *myalbum*, можно скомпилировать другой сегмент *mainprog*, имеющий вид

```
mainprog with stackseg from myalbum
begin
    .
    .
    .
end
```

В сегменте *mainprog* для доступа к стеку, описанному в сегменте *stackseg*, можно использовать процедуры *push* и *pop*, но никак нельзя обратиться к *stack* или *stackptr* идентификаторов. Можно преобразовать структуру данных, представляющих стек, например, в список, переписать процедуры *push* и *pop* и перекомпилировать сегмент *stackseg*. Если старую скомпилированную версию *stackseg*, находящуюся в альбоме, заменить новой, *mainprog*, тогда можно запустить с новым типом стека, но сам этот сегмент останется неизменным.

Аналогично можно описать в отдельном сегменте структуру, представляющую таблицу символов, вместе с процедурами обращения и обновления этой таблицы. Если в список **keep** поместить только имена указанных процедур, то другие сегменты смогут обращаться к таблице символов единственным способом — с помощью специально спроектированных процедур. Далее, если язык реализации обладает соответствующими средствами, можно разрешить одной фазе компилятора обращаться к таблице символов, но не обновлять ее, а другой — обращаться к ней или обновлять. Как и в примере со стеком, мы можем изменять структуру таблицы символов и процедуры обращения к ней и обновления, не изменяя остальной части компилятора.

Естественно ожидать, что при разделении аспектов компиляции вероятность правильной реализации каждого аспекта возрастет. Конечно, проект компилятора в целом также должен быть правильным. Рассмотренные здесь приемы не являются новыми или оригинальными, они просто служат иллюстрацией применения хороших проектных методов структурного программирования к конкретной задаче написания компилятора. Следует также отметить, что повышению надежности компилятора может способствовать и выбор языка реализации.

13.3. ПРОВЕРКА КОМПИЛЯТОРА

Как и другие программы, компилятор не следует считать надежным, пока он не будет тщательно выверен на пробном вводе. Желательно проверять каждый аспект компиляции отдельно по мере разработки компилятора, чтобы устранять большинство возникающих ошибок, а не уже полностью написанный компилятор. Конечно, необходимо проверить и компилятор в целом на предмет устранения каких-либо общих недостатков проектирования. Программы для проверки компилятора должны быть написаны не самими разработчиками, а другими лицами, не знакомыми с деталями реализации. Можно также генерировать иные проверяющие программы с помощью синтаксического анализатора [30].

Упражнения

Эти упражнения охватывают круг вопросов, рассмотренных в гл. 1—13. Многие из них носят обзорный характер, и решения к ним не приводятся.

13.1. Обоснуйте важность определения целей проектирования компилятора на начальном этапе.

13.2. Перечислите факторы, определяющие оптимальное число программистов, которых нужно привлечь к конкретному проекту разработки компилятора.

13.3. Группа разработчиков должна создать ряд компиляторов, предназначенных для работы на одной и той же машине. В какой степени разработчики могут воспользоваться модулями, не зависящими от исходного языка?

13.4. Объясните, как вы можете подойти к решению вопроса выбора языка реализации для компилятора.

13.5. Должен ли разработчик компилятора хорошо знать тот язык, который он реализует?

13.6. Как вам кажется, в правильном ли порядке были рассмотрены в данной книге различные аспекты построения компиляторов? Если вы считаете этот порядок правильным, обоснуйте его, а если нет, предложите свой вариант.

13.7. Предложите темы для дальнейших разработок в области построения компиляторов.

13.8. Какие трудности возникнут при написании компилятора для английского языка?

13.9. Декомпилятор осуществляет трансляцию с объектного кода в исходный. Какие проблемы возникнут при его разработке?

13.10. Рассмотрите аспекты разработки компилятора для Паскаля, Фортрана или любого другого более знакомого вам языка.

РЕШЕНИЯ УПРАЖНЕНИЙ К ГЛ. 1 — 12

Глава 1

1.1. Арифметические выражения генерируют сложение, вычитание, умножение команд и т. д., условные выражения и циклы генерируют команды перехода, а присваивания — команды загрузки и записи в память.

1.2. Потребуется компилятор, написанный в коде 1900. Он осуществляет трансляцию с Алгола 68 в код 1900 (рис. У.1).

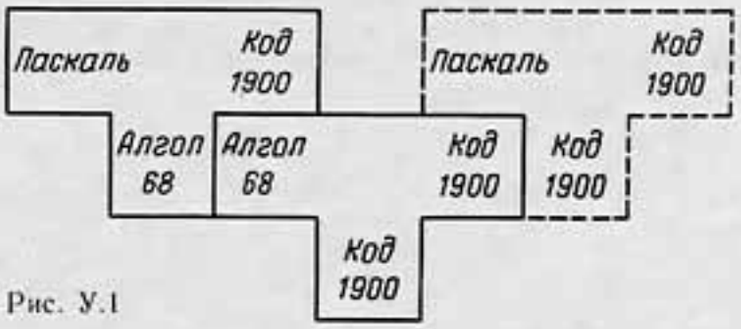


Рис. У.1

1.3. а) Практически необходимы, несмотря на стоимость, так как позволяют проверить, не приводит ли логика программы к ошибкам в индексах.
б) Если логика программы считается правильной, можно опустить, чтобы сделать объектный код меньше и быстрее, но это все же может дать нежелательный эффект.

1.4. Цели 1, 2 и, возможно, 5, 6.

1.5. а) Для стеков и таблиц.
б) Требуются редко.
в) Для модульности.

1.6. 1. Для более четкой диагностики.
2. Потому что в этом случае обычно не имеет значения эффективность работы во время прогона.

1.7. Возможная несовместимость этих двух компиляторов.

1.8. $(a+b\times c)$ обычно представляется бинарным деревом, изображенным на рис. У.2, но если был объявлен приоритет у «+» больше, чем у «x», это дерево будет выглядеть так, как показано на рис. У.3.

1.9. Эффективный объектный код и небольшие объектные программы.

1.10. Построчная компиляция.

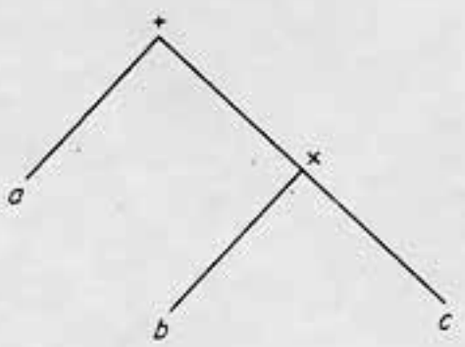


Рис. У.2

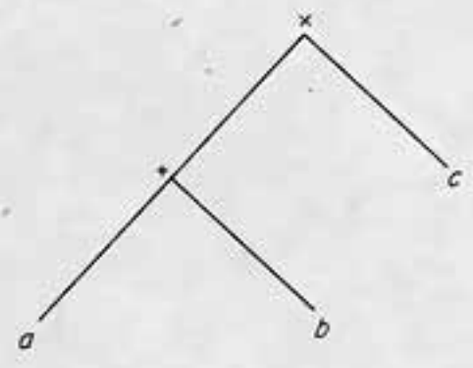


Рис. У.3

Глава 2

2.1. $G_1 = (\{a, b, c\}, \{A, B, C, S\}, P, S)$,

где элементами P являются

$S \to ABC$
$A \to aA$
$A \to \varepsilon$
$B \to bB$
$B \to \varepsilon$
$C \to cC$
$C \to \varepsilon$

$G_2 = (\{a, b, c\}, \{T, C, S\}, P, S)$

где элементами P являются

$S \to TC$
$T \to aTb$
$T \to \varepsilon$
$C \to Cc$
$C \to \varepsilon$

$G_3 = (\{a,b,x,y\}, \{S\}, P, S)$

где элементами P являются

$S \to xSy$
$S \to a$
$S \to b$

2.2. L_1, так как он соответствует регулярному выражению

$$a^*b^*c^*$$

2.3. **Правила разбора:**

$S \to xV$
$S \to yT$
$V \to xV$
$V \to bB$
$V \to b$
$T \to yT$
$T \to bB$
$T \to b$
$B \to bB$
$B \to b$

2.4. $(xx^* | yy^*)bb^*$.

2.5. Предложение

if b **then if** b **then** a **else** a

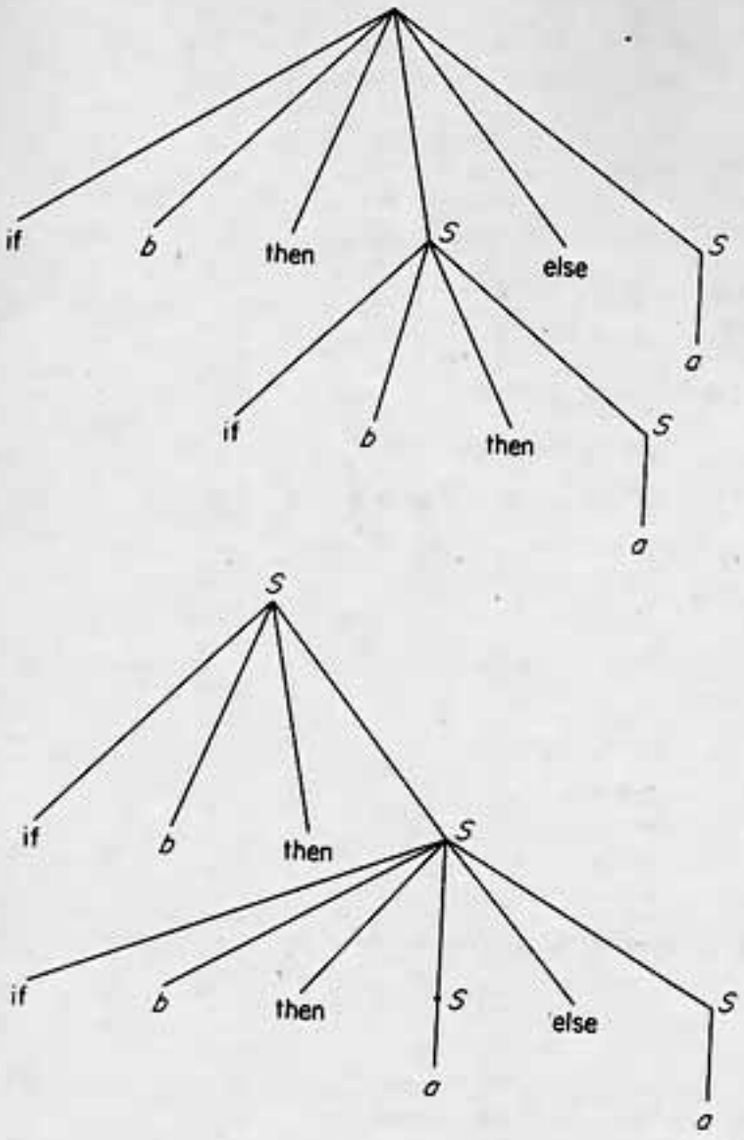
имеет два синтаксических дерева (рис. У.4).

2.6. См. рис. У.5.

2.7. $G = (\{0,1\}, \{S\}, P, S)$

где элементами P являются

$S \to 0S$
$S \to 1S$
$S \to \varepsilon$


if
b
then
S
else
S
a
if
b
then
S
a

S
if
b
then
S
if
b
then
S
else
S
a
a

Рис. У.4

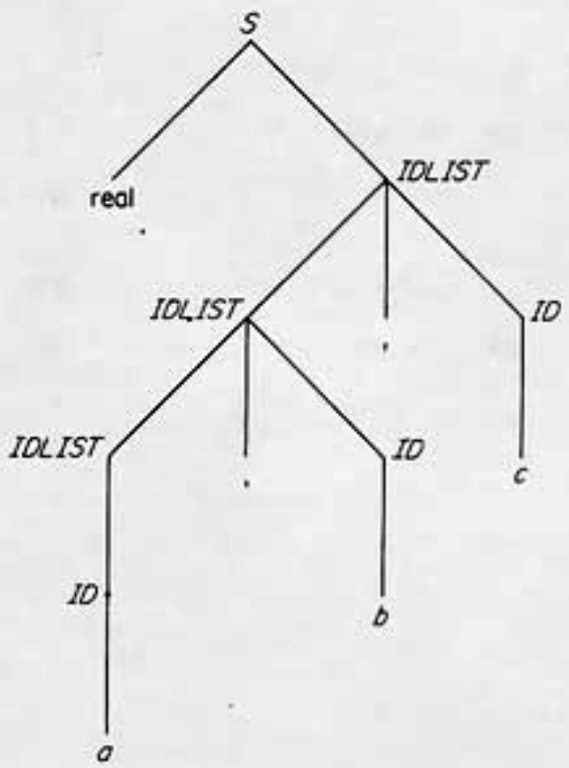

S
real
IDLIST
IDLIST
,
ID
IDLIST
,
ID
c
ID
b
a

Рис. У.5

2.8.

$$
\begin{aligned}
2.8 \quad PROGRAM &\underset{L}{\Rightarrow} \textbf{begin}\ DECS\,;\,STATS\ \textbf{end}\\
&\underset{L}{\Rightarrow} \textbf{begin}\ d\,;\,DECS\,;\,STATS\ \textbf{end}\\
&\underset{L}{\Rightarrow} \textbf{begin}\ d\,;\,d\,;\,STATS\ \textbf{end}\\
&\underset{L}{\Rightarrow} \textbf{begin}\ d\,;\,d\,;\,s\,;\,STATS\ \textbf{end}\\
&\underset{L}{\Rightarrow} \textbf{begin}\ d\,;\,d\,;\,s\,;\,s\ \textbf{end}\\
PROGRAM &\underset{R}{\Rightarrow} \textbf{begin}\ DECS\,;\,STATS\ \textbf{end}\\
&\underset{R}{\Rightarrow} \textbf{begin}\ DECS\,;\,s\,;\,STATS\ \textbf{end}\\
&\underset{R}{\Rightarrow} \textbf{begin}\ DECS\,;\,s\,;\,s\ \textbf{end}\\
&\underset{R}{\Rightarrow} \textbf{begin}\ d\,;\,DECS\,;\,s\,;\,s\ \textbf{end}\\
&\underset{R}{\Rightarrow} \textbf{begin}\ d\,;\,d\,;\,s\,;\,s\ \textbf{end}
\end{aligned}
$$

2.9. а) *буква (буква|цифра|) (буква|цифра|) (буква|цифра|) (буква|цифра|) (буква|цифра|)*.

б) $G = (\{\text{буква, цифра}\}\ \{R, S, T, U, V\}, P, S)$

$S \to$ *буква*
$S \to$ *буква* R
$R \to$ *буква* S
$R \to$ *цифра* S
$R \to$ *буква*
$R \to$ *цифра*
$S \to$ *буква* T
$S \to$ *цифра* T
$S \to$ *буква*
$S \to$ *цифра*
$T \to$ *буква* U
$T \to$ *цифра* U
$T \to$ *буква*
$T \to$ *цифра*
$U \to$ *буква* V
$U \to$ *цифра* V
$U \to$ *буква*
$U \to$ *цифра*
$V \to$ *буква*
$V \to$ *цифра*

2.10. 1. Кобол, Паскаль: при чтении слева направо каждое **'else'** ассоциируется с ближайшим предшествующим **'then'**, еще не ассоциированным с **'else'**.

2. Алгол 68, POP-2: каждое условное выражение заканчивается посредством **fi** или *CLOSE*.

3. Алгол 60: последовательность **'then if** — недопустимая, между ними должно быть **begin**.

Глава 3

3.1.

$S \to 1T$
$T \to 0V$
$V \to 1S$
$S \to 0U$
$U \to 1W$
$W \to 0X$
$X \to 0U$
$W \to 0$
$S \to \varepsilon$

3.2. См. табл. У.0. S — начальное состояние, а S, X — конечные состояния.

Таблица У.0

	Состояния					
	S	T	U	V	W	X
0	U	V			X	U
1	T		W	S		

3.3. См. рис. У.6. S — начальное состояние, а S, F — конечные состояния.

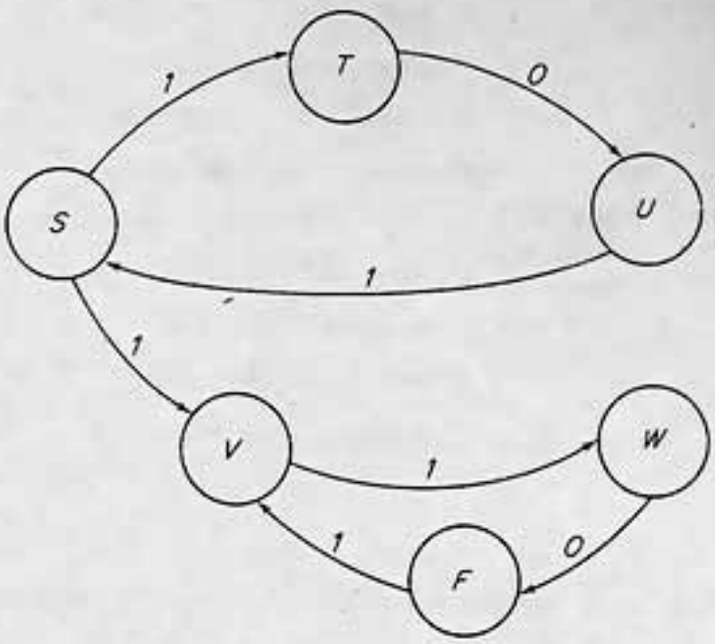


Рис. У.6

3.4. См. рис. У.7. S — начальное состояние, а S, F — конечные состояния.

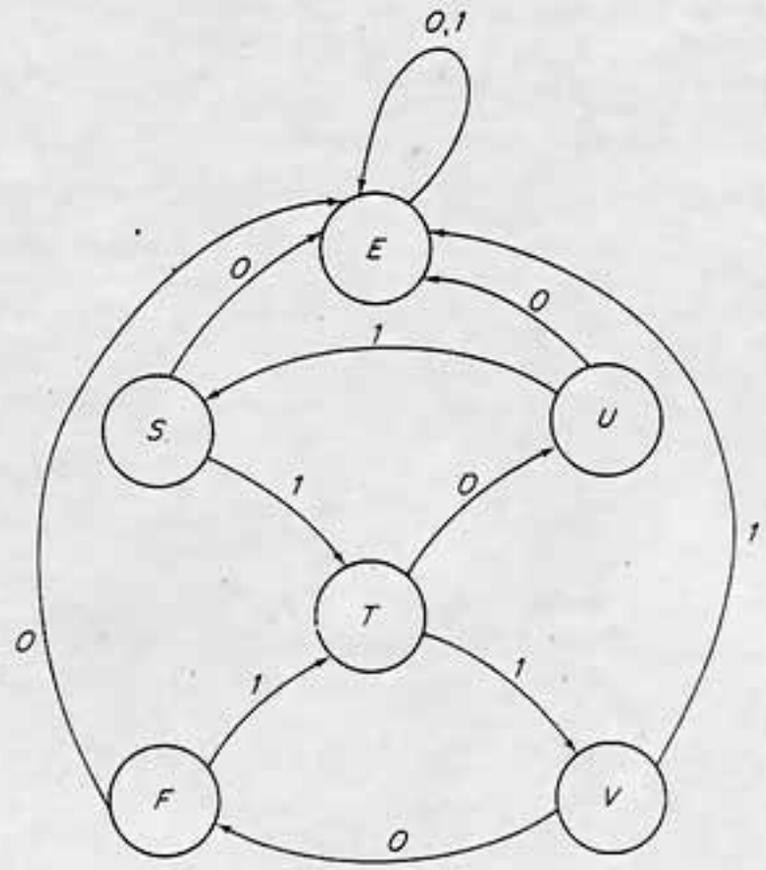


Рис. У.7

3.5. *A* есть начальный символ предложения, а правила разбора следующие:

$A \to 0A$	$C \to 0C$
$A \to 1A$	$C \to 1C$
$A \to 1B$	$C \to 1$
$B \to 1C$	$C \to 0$
$B \to 1$	

Язык состоит из строк, содержащих нули и единицы, причем в строке должно быть не менее двух последовательных единиц.

3.6.

```
proc doubleones = bool:
begin int n, p; bool ok := false;
    read (n);
    while p := n; read (n);
      n = 0 or n = 1 and not ok
    do if p = 1 and n = 1
      then ok := true
      fi
    od;
    while n = 0 or n = 1
    do read (n)
    od;
    ok
end
```

В процедуре предполагается, что строка из нулей и единиц заканчивается каким-либо другим целым числом.

3.7.

```
proc fixed point = void:
 begin char in; read (in);
    bool dig := false;
    if in = "+" or in = "-"
    then read (in)
    fi;
    if digit (in)
    then dig := true;
      read (in)
    fi;
    while digit (in)
    do read (in)
    od;
    if in ≠ "."
    then error
    else read (in)
    fi;
    if digit (in)
    then dig := true; read (in)
    fi;
    while digit (in)
    do read (in)
    od;
    if not dig
    then error
    fi
end
```

3.8. Процедуры *digit* и *error* предполагаются.

```
proc train = void:
begin rollingstock in;
    read (in);
    if in = engine
    then read (in)
    else error
    fi;
    if in = engine
    then read (in)
    fi;
    if in = truck
    then read (in)
    else error
    fi;
    while in = truck
    do read (in)
    od;
    if in ≠ guards van
    then error
    fi
end
```

3.9. Единственно возможным является массив строк.

3.10. Для обозначения этих двух видов можно использовать логическую переменную. Преимущество — исключаются переполнения от рекурсии, недостаток — всякий раз, когда действия компилятора для этих двух видов отличаются, переменную необходимо проверять.

Глава 4

4.1. а) Детерминированный, так как *c* обозначает середину строки, а вторую половину можно согласовать с первой с помощью стека.

б) Недетерминированный, так как при чтении слева направо середину строки определить нельзя, поэтому неизвестно, когда начинать согласование.

в) Детерминированный по причинам, аналогичным изложенным в п. (а).

г) Детерминированный (см. п. (в)).

д) Недетерминированный, поскольку нельзя узнать, снимать ли «*0*» из стека для каждой считанной «*1*» или только для каждой второй считанной «*1*».

4.2. а) Воспользуйтесь леммой подкачки и предположите, что язык контекстно-свободный. Для достаточно большого k строку

$$a^k b^k c^k$$

можно записать в виде

$$uvwxy$$

где $|vwx| \leqslant k$, $|vx| \neq 0$ и для $i \geqslant 0$ строка

$$uv^i wx^i y$$

также находится в языке. Допустим, что $i = 0$, тогда

$$uwy$$

находится в языке. Но uwy образуется из $uvwxy$ путем исключения v и x, и так как $|vwx| \leqslant k$, v и x вместе не могут содержать множества a, b и c (т. е. vwx не может простираться по kb). Поэтому uwy должна все же содержать ka или kc, но не может содержать ka, b и c, поскольку $|vx| \neq 0$. Поэтому она не может включать равное число a, b и c и не находится в языке. Таким образом, возникает противоречие, и язык оказывается не контекстно-свободным.

б) Воспользуйтесь леммой подкачки и предположите, что язык контекстно-свободный. Для достаточно большого простого числа k

можно выразить как $$p = a^k$$

$$uvwxy$$

где $|vwx| \leqslant k$, $|vx| \neq 0$ и для $i \geqslant 0$

$$uv^i wx^i y$$

находится в языке. Такие строки имеют длину $p + i \times l$, где p — длина uwy, а l — длина vx, которая не является нулем. Не все эти длины представляют собой простые числа. Например,

$$p + pl$$

не простое число. Следовательно, здесь имеется противоречие, и язык будет неконтекстно-свободным.

4.3.

```
proc E = void:
begin T;
      G
end;
proc T = void:
begin F;
      U
end
proc F = void:
begin if in = "("
   then read (in); E;
      if in = ")"
      then read (in)
      else error
      fi
   elif identifier (in)
   then read (in)
   else error
   fi
end;
proc U = void:
begin if in = "×"
   then read (in); F; U
   fi
end;
proc G = void:
begin if in = "+"
   then read (in); T; G
   fi
end
```

4.4. а) Это следует из 2.

б) Любая s-грамматика удовлетворяет 1 и 2.

в) Преобразование можно осуществить с помощью правил грамматики, непрерывно заменяя самые левые нетерминалы до тех пор, пока не будет получена q-грамматика.

4.5. S есть начальный символ в каждом случае:

1. $S \to 0S11$
   $S \to a$
2. $S \to 0S$
   $S \to 1T$
   $S \to \varepsilon$
   $T \to 0S$
   $T \to \varepsilon$
3. $S \to 1T$
   $S \to 0U$
   $T \to 0X$
   $U \to 1X$
   $T \to 1UU$
   $U \to 0TT$
   $X \to S$
   $X \to \varepsilon$

4.6. Правила для E и T не позволяют грамматике стать LL(1)-грамматикой

$$E \to TX$$
$$X \to +TX$$
$$X \to \varepsilon$$
$$T \to FY$$
$$Y \to \times FY$$
$$Y \to \varepsilon$$

4.7. Нет, так как x является направляющим символом в этих двух правилах для S.

4.8. Предложение, генерированное LL(1)-грамматикой, не может иметь два левосторонних вывода, так как иначе это означало бы, что два альтернативных правила для какого-либо нетерминала имеют общий направляющий символ.

4.9. См. табл. У.1.

4.10. См. табл. У.2.

Глава 5

5.1. а)
$$S \to aYc$$
$$Y \to Ybb$$
$$Y \to b$$
(S — начальный символ)

б)
$$E \to E+T$$
$$E \to T$$
$$T \to T \times F$$
$$T \to F$$
$$F \to x$$
$$F \to (E)$$
(E — начальный символ)

в)
$$P \to D=S$$
$$P \to S$$
$$D \to {}^*S$$
$$D \to x$$
$$S \to D$$
(P — начальный символ)

5.2. а) Признак LR(1) отсутствует, так как грамматика неоднозначна. Ее можно заменить на

$$E \to E+i$$
$$E \to i$$

б) Это — не LR(1)-грамматика, так как язык нельзя разобрать детерминированно. Соответствующей LR(1)-грамматики не существует.

в) Построив таблицу разбора, можно показать, что грамматика обладает признаком LR(0), а поэтому и признаком LR(1).

5.3. См. табл. У.3. Это — SLR(1)-грамматика.

5.4. Грамматика не обладает признаком LR(1), так как при чтении *d* с *semi* в качестве предварительно просматриваемого символа неизвестно, нужно ли приводить к *DEC-LIST* или нет. Эквивалентной SLR(1)-грамматикой является следующая:

PROGRAM → *begin d semi X end*
X → *d semi X*
X → *s Y*
Y → *semi s Y*
Y → ε

См. табл. У.4.

5.5. См. табл. У.5.

5.6. Грамматика не обладает признаком *LR(1)*, так как язык нельзя разобрать детерминированно. Невозможно, например, на основании проделанного анализа и одного последующего символа определить, требуется ли *01* приводить к *S*.

5.7. а) Нет.
б) Нет.

5.8. Это — не LR(1)-грамматика, так как при чтении «*I*» с последующим символом «:» LR(1)-анализатор не знает, выполнять приведение с помощью правила (4) или нет. Этой проблемы можно избежать, если во время лексического анализа заменить последовательность «:=» одним символом.

5.9. Ошибки могут быть обнаружены на более ранней стадии разбора (в смысле выполненных приведений, а не считанных символов) при меньшем числе действий приведения и большем числе показаний ошибок в таблице разбора.

5.10. Стеки можно объединить, так как нет таких действий в разборе, которые добавляли бы или исключали элемент из одного стека, и не делали этого по отношению к другому. Строго говоря, стек символов не требуется.

5.11. Каждый регулярный язык соответствует детерминированному конечному автомату, из которого можно получить LR(1)-таблицу разбора.

Таблица У.1

	terminals	*jump*	*accept*	*stack*	*return*	*error*
1	{*begin*}	*2*	**false**	**false**	**false**	**true**
2	{*begin*}	*3*	**true**	**false**	**false**	**true**
3	{*d*}	*4*	**true**	**false**	**false**	**true**
4	{*semi*}	*5*	**true**	**false**	**false**	**true**
5	{*d*,*s*}	*7*	**false**	**true**	**false**	**true**
6	{*end*}	*0*	**true**	**false**	**true**	**true**
7	{*d*}	*9*	**false**	**false**	**false**	**false**
8	{*s*}	*12*	**false**	**false**	**false**	**false**
9	{*d*}	*10*	**true**	**false**	**false**	**true**
10	{*semi*}	*11*	**true**	**false**	**false**	**true**
11	{*d*,*s*}	*7*	**false**	**false**	**false**	**true**
12	{*s*}	*13*	**true**	**false**	**false**	**true**
13	{*end*, *semi*}	*14*	**false**	**false**	**false**	**true**
14	{*end*}	*16*	**false**	**false**	**false**	**false**
15	{*semi*}	*17*	**false**	**false**	**false**	**true**
16	{*end*}	*0*	**false**	**false**	**true**	**true**
17	{*semi*}	*18*	**true**	**false**	**false**	**true**
18	{*s*}	*19*	**true**	**false**	**false**	**true**
19	{*end*, *semi*}	*14*	**false**	**false**	**false**	**true**

Таблица У.2

	terminals	*jump*	*accept*	*stack*	*return*	*error*
1	{(,*x*,*y*}	*2*	**false**	**false**	**false**	**true**
2	{(,*x*,*y*}	*4*	**false**	**true**	**false**	**true**
3	{+,⊥,)}	*7*	**false**	**false**	**false**	**true**
4	{(,*x*,*y*}	*5*	**false**	**false**	**false**	**true**
5	{(,*x*,*y*}	*19*	**false**	**true**	**false**	**true**
6	{×,+,⊥,)}	*13*	**false**	**false**	**false**	**true**
7	{+}	*9*	**false**	**false**	**false**	**false**
8	{⊥,)}	*12*	**false**	**false**	**false**	**true**
9	{+}	*10*	**true**	**false**	**false**	**true**
10	{(,*x*,*y*}	*4*	**false**	**true**	**false**	**true**
11	{+,⊥,)}	*7*	**false**	**false**	**false**	**true**
12	{⊥,)}	*0*	**false**	**false**	**true**	**true**
13	{×}	*15*	**false**	**false**	**false**	**false**
14	{+,⊥,)}	*18*	**false**	**false**	**false**	**true**
15	{×}	*16*	**true**	**false**	**false**	**true**
16	{(,*x*,*y*}	*19*	**false**	**true**	**false**	**true**
17	{×,+,⊥,)}	*13*	**false**	**false**	**false**	**true**
18	{+,⊥,)}	*0*	**false**	**false**	**true**	**true**
19	{(}	*22*	**false**	**false**	**false**	**false**
20	{*x*}	*25*	**false**	**false**	**false**	**false**
21	{*y*}	*26*	**false**	**false**	**false**	**true**
22	{(}	*23*	**true**	**false**	**false**	**true**
23	{(,*x*,*y*}	*1*	**false**	**true**	**false**	**true**
24	{)}	*0*	**true**	**false**	**true**	**true**
25	{*x*}	*0*	**true**	**false**	**true**	**true**
26	{*y*}	*0*	**true**	**false**	**true**	**true**

Таблица У.3

	S	E	T	F	'('	')'	'+'	'×'	x y	'⊥'
1	HALT	S2	S3	S4	S5				S6	
2							S7			R1
3						R3	R3	S9		R3
4						R5	R5	R5		R5
5		S11	S3	S4	S5					
6						R7/8	R7/8	R7/8		R7/8
7			S8	S4	S5					
8						R2	R2			R2
9				S10	S5					
10						R4	R4	R4		R4
11						S12				
12						R6	R6	R6		R6

Таблица У.4

	PROGRAM	X	Y	begin	semi	d	s	end	'⊥'
1	HALT			S2					
2						S3			
3					S4				
4		S5				S7	S10		
5								S6	
6									R1
7					S8				
8		S9				S7	S10		
9								R2	
10			S11		S12			R5	
11								R3	
12							S13		
13			S14		S12			R5	
14								R4	

Таблица У.5

	S	*F*	*ITEM*	';'	*a*	*t*	'⊥'
1	*HALT*				*S2*		
2						*S3*	
3		*S4*		*S5*			*R3*
4							*R1*
5			*S6*		*S8*	*S10*	
6		*S7*		*S5*			*R3*
7							*R2*
8						*S9*	
9				*R4*			*R4*
10				*R5*			*R5*

Глава 6

6.1. После идентификатора следует печать. Однако каждый знак операции следует помещать в стек и удалять из него и печатать после *TERM* или *FACT*.

6.2. $E \rightarrow E+T$
$E \rightarrow <A1>E-T$

6.3. Нет.

6.4. После считывания знака операции и перед его передачей синтаксическому анализатору должен проверяться его приоритет.

6.5. 1. Алгебраические выражения.
2. Списки имен и адресов.

6.6. 1. Конечная ширина бумаги позволяет делать отступы (абзацы) только определенной глубины вложения.
2. Разные программисты прибегают к разным условным обозначениям отступов (абзацев).

6.7. С помощью следующей грамматики:

$$\begin{aligned} I &\rightarrow +U \\ &\rightarrow -U<M> \\ U &\rightarrow d<A> \\ &\quad d<A>U \end{aligned}$$

где действиями являются

$$\begin{aligned} &<M> \\ &sum := -sum \\ &<A> \\ &sum := 10 \times sum + d \end{aligned}$$

Здесь *sum* первоначально нулевая, а d — значение последней считанной цифры.

6.8. Нет необходимости иметь таблицы d в виде блочной структуры, рассмотренной в данной главе. Можно использовать массивы.

6.9. 1. Более естественный стиль программирования, позволяющий описывать идентификатор, когда это требуется.
2. В программе будет меньше ошибок, обусловленных недостающими описаниями.

6.10. Каждому вызову действия соответствует дополнительный элемент в таблице разбора.

Глава 7

7.1. 1. Для обеспечения естественного метода, разрешающего применение взаимно рекурсивных процедур и т. п.
2. Чтобы позволить перегружать определенные символы.

7.2. 1. Для уменьшения размеров компилятора.
2. С целью оптимизации объектного кода.
3. Для обеспечения хорошей диагностики во время компиляции.

7.3. Аргумент «за» — все еще доступен оригинальный исходный код. Аргумент «против» — многопроходный компилятор может способствовать выдаче сообщений о синтаксических ошибках на более раннем этапе исходной программы.

7.4. 1. Распознавание заголовков процедур в Алголе 68.
2. Обращение с объявлениями в Паскале.

7.5. Аргумент «за» — помогает в отладке компилятора. Аргумент «против» — чтение и запись между проходами могут быть неэффективными.

7.6. Помогает объяснить последующие сообщения об ошибках.

7.7. Синтаксический анализатор может потерять время за счет неправильного допущения в более раннем проходе.

7.8. 1. Компилятор может воспользоваться таким преимуществом машины, как аппаратурный стек.
2. Возможности машины обусловливают виды оптимизации, выполняемые компилятором.
3. Конструкция генератора кода может зависеть от числа имеющихся регистров.

7.9. 1. Нет зависимости от машины на «переднем крае».
2. Применение абстрактной машины, которая легко реализуется на большинстве ЭВМ.

7.10. 1. Структура компилятора оказывается более тесно связанной с определением исходного языка.
2. Фазы компиляции можно логически разделить.

Глава 8

8.1. а) Преимущество: простая таблица идентификаторов и простые функции хеширования для таблицы символов.
б) Недостаток: нельзя пользоваться мнемоническими идентификаторами.

8.2. Нет, так как могут быть переписаны стандартные типы, принятые по умолчанию.

8.3. Идентификаторы можно было бы заменить последовательными целыми числами, используемыми как индексы к таблице.

8.4. а) Могут быть помещены в таблицу хеширования перед идентификаторами, определяемыми пользователем, на фиксированные позиции, которые не надо было бы рассчитывать для каждой программы заново.
б) Будут помещаться в самый внешний специальный блок.

8.5. Аргумент «за» — исключается агрегирование в самой таблице символов и в таблице может содержаться произвольное число идентификаторов. Аргумент «против» — объем памяти, занимаемый указателями.

8.6. Найдите середину массива и сделайте этот элемент корнем дерева. Левое и правое поддеревья образуются из этих двух половинок массива и т. д.

8.7. Возможность описания взаимно рекурсивных процедур и т. п.

8.8. Аргумент «за» — экономит место в таблице видов. Аргумент «против» — для каждого встреченного нового вида необходимо производить проверку, чтобы убедиться в том, что его еще нет в таблице.

8.9. а) Недопустимое; **real, ref real** слишком тесно связаны.
б) Допустимое; эквивалентно **union (real, char)**.
в) Недопустимое; потребует бесконечный объем памяти.

8.10. Нет. Вид нельзя определить как некоторое число, состоящее из **ref** и/или **proc**, за которыми следует он сам.

Глава 9

9.1. Описывая переменные по возможности локально.

9.2. Процесс выборки значения из стека с помощью *DISPLAY* требует сложения смещения с начальным адресом *DISPLAY* по указателю и прибавления другого смещения. Частично этого можно избежать, если отвести для статических значений отдельный стек в том случае, когда процедуры не нужно реализовывать.

9.3. Преимущество: нет необходимости в дополнительном проходе компилятора, чтобы сделать смещения рабочего стека относительными началу рамки. Недостаток: сложность из-за того, что требуется множество стеков в рабочее время.

9.4. Глобальный объем выделяется на время выполнения программы, а локальный отдается для совместного пользования процедурам. Общий требуемый объем памяти определяется как глобальный объем + локальный объем, нужный каждой процедуре.

9.5. При компиляции блока весь объем памяти для статических идентификаторов может выделяться перед любым статическим рабочим участком памяти. Таким образом, смещения рабочего стека относительно начала рамки будут известны при выделении рабочей памяти. Преимуществом является также возможность контроля динамической рабочей памяти во время прогона.

9.6. Допустим, y — формальный параметр процедуры, вызываемой посредством ссылки или по значению и результату, а x — соответствующий фактический параметр. Присвойте новое значение x внутри процедуры и напечатайте y. Если y вызывается посредством ссылки, то печатается новое значение x, тогда как если y вызывается по значению и результату, печатается предыдущее значение y.

9.7. Вариантам не требуется память для фактического вида/типа.

9.8. Сборка мусора в условиях реального времени не представляется удовлетворительным решением, так как она может вызывать останов программы на неопределенный период. Лучшее решение — использовать счетчик ссылок даже несмотря на то, что в результате выполнение программы несколько замедлится.

9.9. Как отмечалось в этой главе, чем больше обратных указателей, тем больше времени занимает выполнение алгоритма. Предварительный просмотр можно использовать для того, чтобы определить, с какого конца «кучи» алгоритм должен начинать работу.

9.10. Исключается необходимость применения рекурсии или стека.

Глава 10

10.1.

а) $a+b$
$1+c$
$d+e$
$3+f$
2×4

б) $a+b=1$
$1+c=2$
$d+e=3$
$3+f=4$
$2\times4=5$

10.2. Чтобы тройки можно было перегруппировать (например, во время оптимизации), не изменяя их самих.

10.3. Во избежание неоднозначности унарный знак операции в полученной обратной польской записи должен отличаться от соответствующего бинарного знака операции.

10.4. Преимущество — устраняет приоритет знаков операций, позволяя тем самым выполнять трансляцию с помощью простого алгоритма. Недостаток — подход неоптимальный в отношении времени компиляции.

10.5. При компиляции нет необходимости в стеке значений, если нельзя использовать скобки для спецификации приоритетов.

10.6. Во время компиляции все значения должны храниться в стеке до тех пор, пока не будет достигнута правая часть выражения, если не используются скобки.

10.7. Может несколько сократить объем поиска в таблице, но значительно удлинит исходный текст.

10.8. Код, генерированный для оператора *case* в Алголе 68, может иметь вид

```
GOTO L0
SETLABEL L1
first element
GOTO LE
SETLABEL L2
second element
GOTO LE
.
.
.
SETLABEL L0
SWITCH address (I)
GOTO L1
GOTO L2
.
.
.
SETLABEL LE
```

Команда *'SWITCH'* осуществит переход к соответствующему *'GOTO'*, следующему за ней.

10.9. Основные этапы:
1. Проверка нахождения метки в таблице символов — не может сделать переход в блок.
2. Генерация кода для настройки *DISPLAY*, если осуществляется переход из блока.
3. Генерация команды *jump*.

10.10. С точки зрения разработчика компилятора — хорошая, так как ее легко прочитать. Во время трансляции команд промежуточного кода требуется поиск в таблице.

Глава 11

11.1. Преимущества — появится возможность воспользоваться стандартным редактором связей и ассемблером. Недостатки — возможно, менее эффективно; требуется дополнительный проход.

11.2.

```
1. read ((add1, add2));
   convert (add1, 1);
   convert (add2, 2);
   print ((newline, X6, «MOVE 1 1»))
2. read ((label, add1));
   convert (add1, 1);
   print ((newline, X6, «LDX 1 0(1)»,
    newline, X6, «BGT 1», X2, label))
```

Допускается, что все адреса базируются на стеке.

11.3. Генерировать, например,

LDX 1 0(1)

11.4. Во многих языках нельзя печатать *'L'*, за которым сразу же следует целое число.

11.5. Описывая все подобные идентификаторы на самом внешнем уровне. Преимущество — меньше описаний. Недостаток — вероятно, менее удобочитаемый код.

11.6. Когда одно действие во время компиляции генерировало более (например) восьми команд в коде Ассемблера.

11.7. Не все значения могут поместиться в регистр.

11.8. Всякий раз, когда необходимо переписать значение, например, при выходе из блока.

11.9. Придется всегда запоминать две последние команды, генерируемые в коде Ассемблера.

11.10. Проверять каждый безусловный переход — не приводит ли он к другому безусловному переходу.

Глава 12

12.1. Нет.

12.2. Ошибки могут быть обнаружены на более ранней стадии, так как компилятор не путает слова языка с идентификаторами, определяемыми пользователем, что возможно в Фортране и ПЛ/1.

12.3. Способствовать ошибкам могут произвольные правила. Исследования показали, что употребление в качестве разделителя «;», а не терминатора приводит к большему числу ошибок программирования, как и чрезмерное употребление некоторых литер.

12.4. Строгая печать позволяет выдавать более четкие сообщения об ошибках (см. также упражнение 12.2).

12.5. *x*; **end**
интерпретируется как

```
x
semi-colon
dummy statement
end
```

12.6. Открывающие и закрывающие символы комментария не идентичны.

12.7. Нет.

12.8. Нет. Их будет, вероятно, слишком много.

12.9. Чтобы программа прошла быстрее. Полезно, но опасно.

12.10. Необходимо пересылать таблицы и т. п. по мере того, как они будут заполнять первоначально выделенный им объем памяти.

ЛИТЕРАТУРА

1. Aho A. V. and Johnson S. C. LR parsing. — Computing Surveys, 6, 1974, p. 99 — 124.
2. Aho A. V. and Ullman J. D. The Theory of Parsing, Translation, and Compiling. Vol. 1. Parsing, Prentice-Hall, Englewood Cliffs, New York, 1972. Русский перевод: Ахо А., Ульман Д. Теория синтаксического анализа перевода и компиляции/Пер. с англ. В. Н. Агафонова; Под. ред. В. М. Курочкина. — М.: Мир, 1978.
3. Aho A. V. and Ullman J. D. Principles of Compiler Design. Addison-Wesley, Reading, Mass, 1977.
4. Backhouse R. C. Syntax of Programming Languages, Theory and Practice. Prentice Hall, Englewood Cliffs, New York, 1979.
5. Barringer H. and Lindsey C. H. The Manchester ALGOL 68 Compiler, Part 1. Proc. 5th annual iii Conference, Guidel, France, 1977.
6. Bauer F. L. and Eickel J. (Eds) Compiler Construction, an Advanced Course. Springer-Verlag, New York, 1974.
7. Bauer F. L. Historical Remarks on Compiler Construction. — In: F. L. Bauer and J. Eickel (Eds) Compiler Construction, an Advanced Course, Springer-Verlag, New York, 1974, p. 603—621.
8. Bauer H., Becker S., Graham S. and Satterthwaite E. ALGOL W Language Description. Computer Science Department, Stanford University, 1968.
9. Bochmann G. V. Semantic Evaluation from Left to Rigth. CACM, 19, 1973, p. 55—62.
10. Branquart P., Cardinael J.-P., Lewi J., Delescaille J.-P. and Vanbegin M. An Optimised Translation Process and its Application to ALGOL 68, Lecture Notes in Computer Science, 38, Springer-Verlag, New York, 1976.
11. Bratman H. An Alternate Form of the UNCOL Diagram. CACM, 4, 1961, p. 142.
12. Brown P. J. Throw-Away Compiling. — Software Practice and Experience, 6, No. 3, 1976, p. 423—434.
13. Brown P. J. Writing Interactive Compilers and Interpreters, Wiley, Chichester, 1979.
14. Cohen J. and Roth M. S. Analyses of Deterministic Parsing Algorithms. CACM, 21, No. 6, 1978, p. 448 — 458.
15. Dakin R. J. and Poole P. C. A Mixed Code Approach. — Computer Journal, 16, No. 3, 1973, p. 219 — 222.
16. De Remer F. L. Simple LR(k) Grammars. CACM, 14, 1971, p. 453 — 460.
17. De Remer F. L. Review of Formalisms and Notation. — In: F. L. Bauer and J. Eickel (Eds) Compiler Construction, an Advanced Course, Springer-Verlag, New York, 1974, p. 42 — 56.
18. Dijkstra E. W. Cooperating Sequential Processes. — In: F. Genuys (Ed.) Programming Languages, Academic Press, New York, 1968, p. 43 — 112.
19. Foster J. M. A Syntax Improving Device. — Computer Journal, 11, 1968, p. 31 — 34.
20. Gries D. Compiler Construction for Digital Computers. Wiley, New York, 1971. Русский перевод: Грис Д. Конструирование компиляторов для цифровых вычислительных машин/Пер. с англ. Е. Б. Докшицкой, Л. А. Зелениной, Л. Б. Морозовой, В. С. Штаркмана; Под ред. Ю. М. Баяковского, В. С. Штаркмана. — М.: Мир, 1975.
21. Griffiths M. Analyse Déterministe et Compilateurs. Thesis, University of Grenoble, October 1969.
22. Griffiths M. LL(1) Grammars and Analysers. — In: F. L. Bauer and J. Eickel (Eds) Compiler Construction, an Advanced Course, Springer-Verlag, New York, 1974, p. 57—84.
23. Griffiths M. Relationship between the Definition and Implementation of a Language. — In: G. Goos and J. Hartmanis (Eds) Software Engineering.— An Advanced Course (Munich 1972), Lecture Notes in Computer Science, Springer-Verlag, New York, 1975.
24. Hall P. A. V. Computational Structures. MacDonald and Jane's/American Elsevier, 1975. Русский перевод: Холл П. Вычислительные структуры: Введение в нечисленное программирование/Пер. с англ. И. Л. Любимской, Н. Б. Фейгельсон; Под ред. Э. З. Любимского. — М.: Мир, 1978.
25. Hansen W. J. and Boom H. The Report on the Standard Hardware Representation for ALGOL 68. SIGPLAN Notices, 12, No. 5, 1977, p. 80 — 87.
26. Hill U. Special Run-Time Organisation Techniques for ALGOL 68. — In: F. L. Bauer and J. Eickel (Eds) Compiler Construction, an Advanced Course, Springer-Verlag, New York, 1974, p. 222 — 252.
27. Hopcroft J. E. and Ullman J. D. Formal Languages and their Relation to Automata. Addison-Wesley, Reading Mass, 1969.
28. Horning J. J. LR Grammars and Analysers. — In: F. L. Bauer and J. Eickel (Eds) Compiler Construction, an Advanced Course, Springer-Verlag, New York, 1974, a, p. 85 — 108.

29. Horning J. J. Structuring Compiler Development. — In: F. L. Bauer and J. Eickel (Eds) Compiler Construction, an Advanced Course, Springer-Verlag, New York, 1974, b, p. 498 — 524.
30. Housais B. Verification of an ALGOL 68 Implementation. Proc. Strathclyde ALGOL 68 Conference, SIGPLAN Notices, 12, No. 6, 1977, p. 117 — 128.
31. Hunter R. B., McGettrick A. D. and Patel R. R. LL versus LR parsing with illustrations from ALGOL 68. Proc. Strathclyde ALGOL 68 Conference, SIGPLAN Notices, 12, No. 6, 1977, p. 49 — 53.
32. Knuth D. E. On the Translation of Languages from Left to Right. — Information and Control, 8, 1965, p. 607 — 639.
33. Knuth D. E. The Remaining Trouble Spots in ALGOL 60. CACM. 10, No. 10. 1967, p. 611
34. Knuth D. E. Semantics of Context-free Languages. — Maths Systems Theory, 2, No. 2, 1968, a, p. 127 — 145.
35. Knuth D. E. The Art of Computer Programming, 1, Fundamental Algorithms, 1 st ed. Addison-Wesle, Reading, Mass, 1968, b.
36. Knuth D. E. Topdown Syntax Analysis. — Acta Informatica, 1, 1971, p. 79—110.
37. Knuth D. E. The Art of Computer Programming, 1, Fundamental Algorithms, 2nd ed. Addison-Wesley, Reading, Mass, 1973. Русский перевод: Кнут Д. Искусство программирования. Т. 1. — М.: Мир, 1976.
38. Koch W. and Oeters C. Balancing, Canonical Mode Representation and Separate Compilation in the Berlin ALGOL 68 Implementation. Proc. 5th annual iii Conference, Guidel, France, 1977.
39. Koster C. H. A. On Infinite Modes. ALGOL BULLETIN, AB30, 1969, p. 86 — 89.
40. Koster C. H. A. Using the CDL Compiler-Compiler. — In: F. L. Bauer and J. Eickel (Eds) Compiler Construction, an Advanced Course, Springer-Verlag, New York, 1974, p. 366 — 426.
41. Ledgard H. F. Production Systems: or Can we do better than BNF? CACM, 10, No. 2, 1974, p. 94 — 102.
42. Lucas P. and Walk K. On the Formal Description of PL/1. — Annual Review Automatic Programming, 6, No. 3, 1969, p. 105 — 182.
43. McGettrick A. D. ALGOL 68 — a First and Second Course. Cambridge University Press, London, 1978.
44. Marcotty M., Ledgard H. F. and Bochmann G. V. A Sampler of Formal Definitions. — ACM Computing Surveys, 8, No. 2, 1976.
45. Minsky M. L. Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, New York, 1967.
46. Naur P. (Ed.) Revised Report on the Algorithmic Language ALGOL 60. CACM, 6, No. 1, 1963, p. 1 — 17. Русский перевод: Исправленное сообщение об алгоритмическом языке Алгол 60/Дж. В. Бэкус, Ф. Л. Бауэр, Дж. Грин и др.; Под ред. П. Чаура; Пер. с англ. Г. И. Кожухина; Под ред. П. Ершова. — Л.: ЛГУ, 1964. Исправленное сообщение об алгоритмическом языке Алгол 68/Дж. В. Бэкус, Ф. Л. Бауэр, Дж. Грин и др. Под ред. П. Чаура. Пер. с англ. Г. И. Кожухина. Под ред. А. П. Ершова. — Л.: ЛГУ, 1964.
47. Naur P. The Design of the GIER Algol Compiler. — Annual Review Automatic Programming 4, 1964, p. 49 — 85.
48. Poole P. C. Portable and Adaptable Compiler. — In: F. L. Bauer and J. Eickel (Eds) Compiler Construction, an Advanced Course, Springer-Verlag, New York, 1974, p. 427 — 497.
49. Richards M. The Portability of the BCPL Compiler. — Software Practice and Experience, 1, 1971, p. 135 — 146.
50. Rohl J. S. An Introduction to Compiler Writing. MacDonald and Jane's/American Elsevier, 1975.
51. Simonet M. An Attribute Description of a Subset of ALGOL 68. Proc. Strathclyde ALGOL 68 Conference, SIGPLAN Notices, 12, No. 6, 1977, p. 129 — 137.
52. Simpson D. S. Efficient Code Generation for an ALGOL 68 Compiler. Ph. D. Thesis, Department of Computer Science, University of Manchester, 1976.
53. Strasen V. Gaussian Elimination is not Optimal. — Numerische Mathematik, 13, 1969, p. 354 — 356.
54. Uzgalis R., Eggert P. and Book E. Report for System Development Corporation, Sunnyvale, California, 1977.
55. Van Wijngaarden A., Mailloux B. J., Reck J. E. L., Koster C. H. A., Sintzoff M., Lindsey C. H., Meertens L. G. L. T. and Fisker R. G. Revised Report on Algorithmic Language ALGOL 68. — Acta Informatica, 5, Nos 1 — 3, 1975. Русский перевод: Пересмотренное сообщение об Алголе 68/Ред. А. ван Вейнгаарден и др.; Пер. с англ. А. А. Берса; Под ред. А. П. Ершова. — М.: Мир, 1979.
56. Warshall S. A theorem on Boolean matrices. JACM, 9, 1962, p. 11 — 12.
57. Watt D. A. Analysis-oriented Two-level Grammars. Ph. D. Thesis, University of Glasgow, 1974.
58. Williams M. H. Static Semantic features of ALGOL 60 and BASIC. — Computer Journal, 61, No. 3, 1978, p. 234 — 242.

ОГЛАВЛЕНИЕ

Р. Хантер
Проектирование и конструирование компиляторов

Книга одобрена на заседании секции редсовета по электронной обработке данных в экономике 2 марта 1982 г.

Зав. редакцией **А. В. Павлюков.** Редактор **Н. К. Логинова.** Мл. редактор **О. Б. Степанченко.** Техн. редактор **И. В. Завгороднян.** Корректоры **Г. В. Хлопцева, Н. П. Сперанская, И. П. Елкина.** Худож.-редактор **Е. К. Самойлов.** Переплет художника **Н. Пашуры.**

ИБ № 1402

Сдано в набор 12.08.83. Подписано в печать 16.01.84. Формат 70×100¹/₁₆. Бум. офсетная № 2. Гарнитура «Литературная». Печать офсетная. Усл. п. л. 18,85. Уч.-изд. л. 15,83. Усл. кр.-отт. 18,85. Тираж 17 000 экз. Заказ 1733. Цена 1 р. 30 к.

Издательство «Финансы и статистика», 101000, Москва, ул. Чернышёвского, 7.

Московская типография № 4 Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 129041, Москва, Б. Переяславская ул., д. 46