Основы конструирования компиляторов

В.А.Серебряков, М.П.Галочкин

Предисловие

Предлагаемая вниманию читателя книга основана на курсе лекций, прочитанных на факультете вычислительной математики и кибернетики Московского государственного университета и факультете управления и прикладной математики Московского физико-технического института в 1991-1999 гг. Авторы надеются, что издание книги восполнит существенный пробел в литературе на русском языке по разработке компиляторов.

Содержание книги представляет собой “классические” разделы предмета: лексический и синтаксический анализ, организация памяти компилятора (таблицы символов) и периода исполнения (магазина), генерация кода. Рассматриваются некоторые средства автоматизации процесса разработки трансляторов, такие как LEX, YACC, СУПЕР, методы генерации оптимального кода. Сделана попытка на протяжении всего изложения провести единую “атрибутную” точку зрения на процесс разработки компилятора. В книге не затрагиваются чрезвычайно важные вопросы глобальной оптимизации и разработки компиляторов для машин с параллельной архитектурой. Авторы надеются восполнить эти пробелы в будущем.

Книга будет полезной как студентам и аспирантам программистских специальностей, так и профессионалам в этих областях.

Авторы с благодарностью примут все конструктивные замечания по содержанию книги.

В.А.Серебряков, *serebr@ccas.ru*
М.П.Галочкин, *galoch@ccas.ru*

Оглавление

Глава 1

Введение

1.1 Место компилятора в программном обеспечении

Компиляторы составляют существенную часть программного обеспечения ЭВМ. Это связано с тем, что языки высокого уровня стали основным средством разработки программ. Только очень незначительная часть программного обеспечения, требующая особой эффективности, программируется с помощью ассемблеров. В настоящее время распространено довольно много языков программирования. Наряду с традиционными языками, такими, как Фортран, широкое распространение получили так называемые "универсальные" языки (Паскаль, Си, Модула-2, Ада и другие), а также некоторые специализированные (например, язык обработки списочных структур Лисп). Кроме того, большое распространение получили языки, связанные с узкими предметными областями, такие, как входные языки пакетов прикладных программ.

Для некоторых языков имеется довольно много реализаций. Например, реализаций Паскаля, Модулы-2 или Си для ЭВМ типа IBM PC на рынке десятки.

С другой стороны, постоянно растущая потребность в новых компиляторах связана с бурным развитием архитектур ЭВМ. Это развитие идет по различным направлениям. Совершенствуются старые архитектуры как в концептуальном отношении, так и по отдельным, конкретным линиям. Это можно проиллюстрировать на примере микропроцессора Intel-80X86. Последовательные версии этого микропроцессора 8086, 80186, 80286, 80386, 80486, 80586 отличаются не только техническими характеристиками, но и, что более важно, новыми возможностями и, значит, изменением (расширением) системы команд. Естественно, это требует новых компиляторов (или модификации старых). То же можно сказать о микропроцессорах Motorola 68010, 68020, 68030, 68040.

В рамках традиционных последовательных машин возникает боль-

шое число различных направлений архитектур. Примерами могут служить архитектуры CISC, RISC. Такие ведущие фирмы, как Intel, Motorola, Sun, начинают переходить на выпуск машин с RISC-архитектурами. Естественно, для каждой новой системы команд требуется полный набор новых компиляторов с распространенных языков.

Наконец, бурно развиваются различные параллельные архитектуры. Среди них отметим векторные, многопроцессорные, с широким командным словом (вариантом которых являются суперскалярные ЭВМ). На рынке уже имеются десятки типов ЭВМ с параллельной архитектурой, начиная от супер-ЭВМ (Cray, CDC и другие), через рабочие станции (например, IBM RS/6000) и кончая персональными (например, на основе микропроцессора I-860). Естественно, для каждой из машин создаются новые компиляторы для многих языков программирования. Здесь необходимо также отметить, что новые архитектуры требуют разработки совершенно новых подходов к созданию компиляторов, так что наряду с собственно разработкой компиляторов ведется и большая научная работа по созданию новых методов трансляции.

1.2 Структура компилятора

Обобщенная структура компилятора и основные фазы компиляции показаны на рис. 1.1.

На фазе лексического анализа входная программа, представляющая собой поток литер, разбивается на лексемы – слова в соответствии с определениями языка. Основными формализмами, лежащим в основе реализации лексических анализаторов, являются конечные автоматы и регулярные выражения. Лексический анализатор может работать в двух основных режимах: либо как подпрограмма, вызываемая синтаксическим анализатором для получения очередной лексемы, либо как полный проход, результатом которого является файл лексем.

В процессе выделения лексем лексический анализатор может как самостоятельно строить таблицы объектов (идентификаторов, строк, чисел и т.д.), так и выдавать значения для каждой лексемы при очередном к нему обращении. В этом случае таблицы объектов строятся в последующих фазах (например, в процессе синтаксического анализа).

На этапе лексического анализа обнаруживаются некоторые (простейшие) ошибки (недопустимые символы, неправильная запись чисел, идентификаторов и др.).

Основная задача синтаксического анализа – разбор структуры программы. Как правило, под структурой понимается дерево, соответствующее разбору в контекстно-свободной грамматике языка. В настоящее время чаще всего используется либо LL(1)-анализ (и его вариант – рекурсивный спуск), либо LR(1)-анализ и его варианты (LR(0), SLR(1), LALR(1) и другие). Рекурсивный спуск чаще используется при ручном программировании синтаксического анализатора, LR(1) – при исполь-


Лексический анализ
Конечные автоматы
Диагностика
Поток лексем + таблицы объектов
Синтаксический анализ
КС-грамматики
Диагностика
Дерево разбора + таблицы объектов
Контекстный анализ
Атрибутные грамматики
Диагностика
Атрибутированное дерево + таблицы объектов
Генерация промежуточного представления
СУ-трансляция
Промежуточная форма (префиксная, постфиксная, тройки и др.)
Оптимизация
Потоковый анализ
Промежуточная форма (ориентированный граф)
Генерация кода
Таблицы решений, динамическое программирование
Объектный модуль

Рис. 1.1:

зовании систем автоматического построения синтаксических анализаторов.

Результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицы объектов. В процессе синтаксического анализа также обнаруживаются ошибки, связанные со структурой программы.

На этапе контекстного анализа выявляются зависимости между частями программы, которые не могут быть описаны контекстно-свободным синтаксисом. Это в основном связи “описание-использование”, в частности, анализ типов объектов, анализ областей видимости, соответствие параметров, метки и другие. В процессе контекстного анализа таблицы объектов пополняются информацией об описаниях (свойствах) объектов.

Основным формализмом, использующимся при контекстном анализе, является аппарат атрибутных грамматик. Результатом контекстного анализа является атрибутированное дерево программы. Информация об объектах может быть как рассредоточена в самом дереве, так и сосредоточена в отдельных таблицах объектов. В процессе контекстного анализа также могут быть обнаружены ошибки, связанные с неправильным использованием объектов.

Затем программа может быть переведена во внутреннее представление. Это делается для целей оптимизации и/или удобства генерации кода. Еще одной целью преобразования программы во внутреннее представление является желание иметь переносимый компилятор. Тогда только последняя фаза (генерация кода) является машинно-зависимой. В качестве внутреннего представления может использоваться префиксная или постфиксная запись, ориентированный граф, тройки, четверки и другие.

Фаз оптимизации может быть несколько. Оптимизации обычно делят на машинно-зависимые и машинно-независимые, локальные и глобальные. Часть машинно-зависимой оптимизации выполняется на фазе генерации кода. Глобальная оптимизация пытается принять во внимание структуру всей программы, локальная – только небольших ее фрагментов. Глобальная оптимизация основывается на глобальном потоковом анализе, который выполняется на графе программы и представляет по существу преобразование этого графа. При этом могут учитываться такие свойства программы, как межпроцедурный анализ, межмодульный анализ, анализ областей жизни переменных и т.д.

Наконец, генерация кода – последняя фаза трансляции. Результатом ее является либо ассемблерный модуль, либо объектный (или загрузочный) модуль. В процессе генерации кода могут выполняться некоторые локальные оптимизации, такие как распределение регистров, выбор длинных или коротких переходов, учет стоимости команд при выборе конкретной последовательности команд. Для генерации кода разработаны различные методы, такие как таблицы решений, сопоставление образцов, включающее динамическое программирование, различ-

ные синтаксические методы.

Конечно, те или иные фазы транслятора могут либо отсутствовать совсем, либо объединяться. В простейшем случае однопроходного транслятора нет явной фазы генерации промежуточного представления и оптимизации, остальные фазы объединены в одну, причем нет и явно построенного синтаксического дерева.

Глава 2

Языки и их представление

2.1 Алфавиты, цепочки и языки

Алфавит, или *словарь* – это конечное множество *символов*. Для обозначения символов мы будем пользоваться цифрами, латинскими буквами и специальными литерами типа $\#$, $\$$.

Пусть V – алфавит. *Цепочка в алфавите* V – это любая строка конечной длины, составленная из символов алфавита V. Синонимом цепочки являются *предложение*, *строка* и *слово*. Пустая цепочка (обозначается e) – это цепочка, в которую не входит ни один символ.

Конкатенацией цепочек x и y называется цепочка xy. Заметим, что $xe = ex = x$ для любой цепочки x.

Пусть x, y, z – произвольные цепочки в некотором алфавите. Цепочка y называется *подцепочкой* цепочки xyz. Цепочки x и y называются, соответственно, *префиксом* и *суффиксом* цепочки xy. Заметим, что любой префикс или суффикс цепочки является подцепочкой этой цепочки. Кроме того, пустая цепочка является префиксом, суффиксом и подцепочкой для любой цепочки.

Пример 2.1. Для цепочки $abbba$ префиксом является любая цепочка из множества $L_1 = \{e, a, ab, abb, abbb, abbba\}$, суффиксом является любая цепочка из множества $L_2 = \{e, a, ba, bba, bbba, abbba\}$, подцепочкой является любая цепочка из множества $L_1 \cup L_2$.

Длиной цепочки w (обозначается $|w|$) называется число символов в ней. Например, $|abababa| = 7$, а $|e| = 0$.

Язык в алфавите V – это некоторое множество цепочек в алфавите V.

Пример 2.2. Пусть дан алфавит $V = \{a, b\}$. Вот некоторые языки в алфавите V:

а) $L_1 = \varnothing$ – пустой язык;

б) $L_2 = \{e\}$ – язык, содержащий только пустую цепочку
(заметим, что L_1 и L_2 – различные языки);

в) $L_3 = \{e, a, b, aa, ab, ba, bb\}$ – язык, содержащий цепочки из a и b, длина которых не превосходит 2;

г) L_4 – язык, включающий всевозможные цепочки из a и b, содержащие четное число a и четное число b;

д) $L_5 = \{a^{n^2} | n > 0\}$ – язык цепочек из a, длины которых представляют собой квадраты натуральных чисел.

Два последних языка содержат бесконечное число цепочек.

Введем обозначение V^* для множества всех цепочек в алфавите V, включая пустую цепочку. Каждый язык в алфавите V является подмножеством V^*. Для обозначения множества всех цепочек в алфавите V, кроме пустой цепочки, будем использовать V^+.

Пример 2.3. Пусть $V = \{0, 1\}$. Тогда $V^* = \{e, 0, 1, 00, 01, 10, 11, 000, ...\}$, $V^+ = \{0, 1, 00, 01, 10, 11, 000, ...\}$.

Введем некоторые операции над языками.

Пусть L_1 и L_2 – языки в алфавите V. *Конкатенацией* языков L_1 и L_2 называется язык $L_1L_2 = \{xy | x \in L_1, y \in L_2\}$.

Пусть L – язык в алфавите V. *Итерацией* языка L называется язык L^*, определяемый следующим образом:

(1) $L^0 = \{e\}$;

(2) $L^n = LL^{n-1}$, $n \geqslant 1$;

(3) $L^* = \bigcup\limits_{n=0}^{\infty} L^n$.

Пример 2.4. Пусть $L_1 = \{aa, bb\}$ и $L_2 = \{e, a, bb\}$. Тогда
$L_1L_2 = \{aa, bb, aaa, bba, aabb, bbbb\}$, и
$L_1^* = \{e, aa, bb, aaaa, aabb, bbaa, bbbb, aaaaaa, ...\}$.

Большинство языков, представляющих интерес, содержат бесконечное число цепочек. При этом возникают три важных вопроса.

Во-первых, как *представить* язык (т.е. специфицировать входящие в него цепочки)? Если язык содержит только конечное множество цепочек, ответ прост. Можно просто перечислить его цепочки. Если язык бесконечен, необходимо найти для него конечное представление. Это конечное представление, в свою очередь, будет строкой символов над некоторым алфавитом вместе с некоторой интерпретацией, связывающей это представление с языком.

Во-вторых, для любого ли языка существует конечное представление? Можно предположить, что ответ отрицателен. Мы увидим, что множество всех цепочек над алфавитом счетно. Язык – это любое подмножество цепочек. Из теории множеств известно, что множество всех подмножеств счетного множества несчетно. Хотя мы и не дали строгого определения того, что является конечным представлением, интуитивно ясно, что любое разумное определение конечного представления ведет только к счетному множеству конечных представлений, поскольку нужно иметь возможность записать такое конечное представление в виде строки символов конечной длины. Поэтому языков значительно больше, чем конечных представлений.

В-третьих, можно спросить, какова *структура* тех классов языков, для которых существует конечное представление?

2.2 Представление языков

Процедура – это конечная последовательность инструкций, которые могут быть механически выполнены. Примером может служить машинная программа. Процедура, которая всегда заканчивается, называется *алгоритмом*.

Один из способов представления языка – дать алгоритм, определяющий, принадлежит ли цепочка языку. Более общий способ состоит в том, чтобы дать процедуру, которая останавливается с ответом "да" для цепочек, принадлежащих языку, и либо останавливается с ответом "нет", либо вообще не останавливается для цепочек, не принадлежащих языку. Говорят, что такая процедура или алгоритм *распознает* язык.

Такой метод представляет язык с точки зрения распознавания. Язык можно также представить методом порождения. А именно, можно дать процедуру, которая систематически *порождает* в определенном порядке цепочки языка.

Если мы можем распознать цепочки языка над алфавитом V либо с помощью процедуры, либо с помощью алгоритма, то мы можем и генерировать язык, поскольку мы можем систематически генерировать все цепочки из V^*, проверять каждую цепочку на принадлежность языку и выдавать список только цепочек языка. Но если процедура не всегда заканчивается при проверке цепочки, мы не сдвинемся дальше первой цепочки, на которой процедура не заканчивается. Эту проблему можно обойти, организовав проверку таким образом, чтобы процедура никогда не продолжала проверять одну цепочку бесконечно. Для этого введем следующую конструкцию.

Предположим, что V имеет p символов. Мы можем рассматривать цепочки из V^* как числа, представленные в базисе p, плюс пустая цепочка e. Можно занумеровать цепочки в порядке возрастания длины и в "числовом" порядке для цепочек одинаковой длины. Такая нумерация для цепочек языка $\{a, b, c\}^*$ приведена на рис. 2.1, а.

Пусть P – процедура для проверки принадлежности цепочки языку L. Предположим, что P может быть представлена дискретными шагами, так что имеет смысл говорить об i-ом шаге процедуры для любой данной цепочки. Прежде чем дать процедуру перечисления цепочек языка L, дадим процедуру нумерации пар положительных чисел.

Все упорядоченные пары положительных чисел (x, y) можно отобразить на множество положительных чисел следующей формулой:

$$z = (x + y - 1)(x + y - 2)/2 + y$$

Пары целых положительных чисел можно упорядочить в соответствии со значением z (рис. 2.1, б).

1	e
2	a
3	b
4	c
5	aa
6	ab
7	ac
8	ba
9	bb
...	...

а

x \ y	1	2	3	4	5
1	1	3	6	10	15
2	2	5	9	14	
3	4	8	13		
4	7	12			
5	11			$z(x, y)$	

б

Рис. 2.1:

Теперь можно дать процедуру перечисления цепочек L. Нумеруем упорядоченные пары целых положительных чисел – (1,1), (2,1), (1,2), (3,1), (2,2), ... При нумерации пары (i, j) генерируем i-ю цепочку из V^* и применяем к цепочке первые j шагов процедуры P. Как только мы определили, что сгенерированная цепочка принадлежит L, добавляем цепочку к списку элементов L. Если цепочка i принадлежит L, это будет определено P за j шагов для некоторого конечного j. При перечислении (i, j) будет сгенерирована цепочку с номером i. Легко видеть, что эта процедура перечисляет все цепочки L.

Если мы имеем процедуру генерации цепочек языка, то мы всегда можем построить процедуру распознавания предложений языка, но не всегда алгоритм. Для определения того, принадлежит ли x языку L, просто нумеруем предложения L и сравниваем x с каждым предложением. Если сгенерировано x, процедура останавливается, распознав, что x принадлежит L. Конечно, если x не принадлежит L, процедура никогда не закончится.

Язык, предложения которого могут быть сгенерированы процедурой, называется *рекурсивно перечислимым*. Язык рекурсивно перечислим, если имеется процедура, распознающая предложения языка. Говорят,

что язык *рекурсивен*, если существует алгоритм для распознавания языка. Класс рекурсивных языков является собственным подмножеством класса рекурсивно перечислимых языков. Мало того, существуют языки, не являющиеся даже рекурсивно перечислимыми.

2.3 Грамматики

2.3.1 Формальное определение грамматики

Для нас наибольший интерес представляет одна из систем генерации языков – *грамматики*. Понятие грамматики изначально было формализовано лингвистами при изучении естественных языков. Предполагалось, что это может помочь при их автоматической трансляции. Однако, наилучшие результаты в этом направлении достигнуты при описании не естественных языков, а языков программирования. Примером может служить способ описания синтаксиса языков программирования при помощи БНФ – формы Бэкуса-Наура.

Определение. Грамматика – это четверка $G = (N, T, P, S)$, где

(1) N – алфавит *нетерминальных символов*;

(2) T – алфавит *терминальных символов*, $N \cap T = \varnothing$;

(3) P – конечное множество *правил* вида $\alpha \to \beta$, где $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$;

(4) $S \in N$ – *начальный символ* (или *аксиома*) грамматики.

Мы будем использовать большие латинские буквы для обозначения нетерминальных символов, малые латинские буквы из начала алфавита для обозначения терминальных символов, малые латинские буквы из конца алфавита для обозначения цепочек из T^* и, наконец, малые греческие буквы для обозначения цепочек из $(N \cup T)^*$.

Будем использовать также сокращенную запись $A \to \alpha_1|\alpha_2| \ldots |\alpha_n$ для обозначения группы правил $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_n$.

Определим на множестве $(N \cup T)^*$ бинарное отношение *выводимости* $\Rightarrow$ следующим образом: если $\delta \to \gamma \in P$, то $\alpha\delta\beta \Rightarrow \alpha\gamma\beta$ для всех $\alpha, \beta \in (N \cup T)^*$. Если $\alpha_1 \Rightarrow \alpha_2$, то говорят, что цепочка α_2 *непосредственно выводима* из α_1.

Мы будем использовать также рефлексивно-транзитивное и транзитивное замыкания отношения $\Rightarrow$, а также его степень $k \geqslant 0$ (обозначаемые соответственно $\Rightarrow^*$, $\Rightarrow^+$ и $\Rightarrow^k$). Если $\alpha_1 \Rightarrow^* \alpha_2$ ($\alpha_1 \Rightarrow^+ \alpha_2$, $\alpha_1 \Rightarrow^k \alpha_2$), то говорят, что цепочка α_2 *выводима* (*нетривиально выводима*, *выводима за k шагов*) из α_1.

Если $\alpha \Rightarrow^k \beta$ ($k \geqslant 0$), то существует последовательность шагов

$$\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{k-1} \Rightarrow \gamma_k$$

где $\alpha = \gamma_0$ и $\beta = \gamma_k$. Последовательность цепочек $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k$ в этом случае называют *выводом* β из α.

Сентенциальной формой грамматики G называется цепочка, выводимая из ее начального символа.

Языком, порождаемым грамматикой G (обозначается $L(G)$), называется множество всех ее терминальных сентенциальных форм, т.е.

$$L(G) = \{w | w \in T^*, S \Rightarrow^+ w\}$$

Грамматики G_1 и G_2 называются *эквивалентными*, если они порождают один и тот же язык, т.е. $L(G_1) = L(G_2)$.

Пример 2.5. Грамматика $G = (\{S, B, C\}, \{a, b, c\}, P, S)$, где $P = \{S \to aSBC, S \to aBC, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc\}$, порождает язык $L(G) = \{a^n b^n c^n | n > 0\}$.

Действительно, применяем $n-1$ раз правило 1 и получаем $a^{n-1}S(BC)^{n-1}$, затем один раз правило 2 и получаем $a^n(BC)^n$, затем $n(n-1)/2$ раз правило 3 и получаем $a^n B^n C^n$.

Затем используем правило 4 и получаем $a^n b B^{n-1} C^n$. Затем применяем $n-1$ раз правило 5 и получаем $a^n b^n C^n$. Затем применяем правило 6 и $n-1$ раз правило 7 и получаем $a^n b^n c^n$. Можно показать, что язык $L(G)$ состоит из цепочек только такого вида.

Пример 2.6. Рассмотрим грамматику $G = (\{S\}, \{0, 1\}, \{S \to 0S1, S \to 01\}, S)$. Легко видеть, что цепочка $000111 \in L(G)$, так как существует вывод

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000111$$

Нетрудно показать, что грамматика порождает язык $L(G) = \{0^n 1^n | n > 0\}$.

Пример 2.7. Рассмотрим грамматику $G = (\{S, A\}, \{0, 1\}, \{S \to 0S, S \to 0A, A \to 1A, A \to 1\}, S)$. Нетрудно показать, что грамматика порождает язык $L(G) = \{0^n 1^m | n, m > 0\}$.

2.3.2 Типы грамматик и их свойства

Рассмотрим классификацию грамматик (предложенную Н.Хомским), основанную на виде их правил.

Определение. Пусть дана грамматика $G = (N, T, P, S)$. Тогда

(1) если правила грамматики не удовлетворяют никаким ограничениям, то ее называют грамматикой типа 0, или грамматикой *без ограничений*.

(2) если

   а) каждое правило грамматики, кроме $S \to e$, имеет вид $\alpha \to \beta$, где $|\alpha| \leqslant |\beta|$, и

б) в том случае, когда $S \to e \in P$, символ S не встречается в правых частях правил,

то грамматику называют грамматикой типа 1, или *неукорачивающей*.

(3) если каждое правило грамматики имеет вид $A \to \beta$, где $A \in N$, $\beta \in (N \cup T)^*$, то ее называют грамматикой типа 2, или *контекстно-свободной* (КС-грамматикой).

(4) если каждое правило грамматики имеет вид либо $A \to xB$, либо $A \to x$, где $A, B \in N$, $x \in T^*$ то ее называют грамматикой типа 3, или *праволинейной*.

Легко видеть, что грамматика в примере 2.5 – неукорачивающая, в примере 2.6 – контекстно-свободная, в примере 2.7 – праволинейная.

Язык, порождаемый грамматикой типа i, называют языком типа i. Язык типа 0 называют также языком без ограничений, язык типа 1 – контекстно-зависимым (КЗ), язык типа 2 – контекстно-свободным (КС), язык типа 3 – праволинейным.

Теорема 2.1. *Каждый контекстно-свободный язык может быть порожден неукорачивающей грамматикой.*

Доказательство. Пусть L – контекстно-свободный язык. Тогда существует контекстно-свободная грамматика $G = (N, T, P, S)$, порождающая L.

Построим новую грамматику $G' = (N', T, P', S')$ следующим образом:

1. Если в P есть правило вида $A \to \alpha_0 B_1 \alpha_1 \ldots B_k \alpha_k$, где $k \geqslant 0$, $B_i \Rightarrow^+ e$ для $1 \leqslant i \leqslant k$, и ни из одной цепочки α_j $(0 \leqslant j \leqslant k)$ не выводится e, то включить в P' все правила (кроме $A \to e$) вида

$$A \to \alpha_0 X_1 \alpha_1 \ldots X_k \alpha_k$$

где X_i – это либо B_i, либо e.

2. Если $S \Rightarrow^+ e$, то включить в P' правила $S' \to S$, $S' \to e$ и положить $N' = N \cup \{S'\}$. В противном случае положить $N' = N$ и $S' = S$.

Легко видеть, что G' – неукорачивающая грамматика. Можно показать по индукции, что $L(G') = L(G)$.

■

Пусть K_i – класс всех языков типа i. Доказано, что справедливо следующее (строгое) включение: $K_3 \subset K_2 \subset K_1 \subset K_0$.

Заметим, что если язык порождается некоторой грамматикой, это не означает, что он не может быть порожден грамматикой с более сильными ограничениями на правила. Приводимый ниже пример иллюстрирует этот факт.

Пример 2.8. Рассмотрим грамматику $G = (\{S, A, B\}, \{0, 1\}, \{S \to AB, A \to 0A, A \to 0, B \to 1B, B \to 1\}, S)$. Эта грамматика является контекстно-свободной. Легко показать, что $L(G) = \{0^n 1^m | n, m > 0\}$. Однако, в примере 2.7 приведена праволинейная грамматика, порождающая тот же язык.

Покажем что существует алгоритм, позволяющий для произвольного КЗ-языка L в алфавите T, и произвольной цепочки $w \in T^*$ определить, принадлежит ли w языку L.

Теорема 2.2. *Каждый контекстно-зависимый язык является рекурсивным языком.*

Доказательство. Пусть L – контекстно-зависимый язык. Тогда существует некоторая неукорачивающая грамматика $G = (N, T, P, S)$, порождающая L.

Пусть $w \in T^*$ и $|w| = n$. Если $n = 0$, т.е. $w = e$, то принадлежность $w \in L$ проверяется тривиальным образом. Так что будем предполагать, что $n > 0$.

Определим множество T_m как множество строк $u \in (N \cup T)^+$ длины не более n таких, что вывод $S \Rightarrow^* u$ имеет не более m шагов. Ясно, что $T_0 = \{S\}$.

Легко показать, что T_m можно получить из T_{m-1} просматривая, какие строки с длиной, меньшей или равной n можно вывести из строк из T_{m-1} применением одного правила, т.е.

$$T_m = T_{m-1} \cup \{u \mid v \Rightarrow u \text{ для некоторого } v \in T_{m-1}, \text{ где } |u| \leqslant n\}.$$

Если $S \Rightarrow^* u$ и $|u| \leqslant n$, то $u \in T_m$ для некоторого m. Если из S не выводится u или $|u| > n$, то u не принадлежит T_m ни для какого m.

Очевидно, что $T_m \supseteq T_{m-1}$ для всех $m \geqslant 1$. Поскольку T_m зависит только от T_{m-1}, если $T_m = T_{m-1}$, то $T_m = T_{m+1} = T_{m+2} = \ldots$. Процедура будет вычислять $T_1, T_2, T_3, \ldots$ пока для некоторого m не окажется $T_m = T_{m-1}$. Если w не принадлежит T_m, то не принадлежит и $L(G)$, поскольку для $j > m$ выполнено $T_j = T_m$. Если $w \in T_m$, то $S \Rightarrow^* w$.

Покажем, что существует такое m, что $T_m = T_{m-1}$. Поскольку для каждого $i \geqslant 1$ справедливо $T_i \supseteq T_{i-1}$, то если $T_i \neq T_{i-1}$, то число элементов в T_i по крайней мере на 1 больше, чем в T_{i-1}. Пусть $|N \cup T| = k$. Тогда число строк в $(N \cup T)^+$ длины меньшей или равной n равно $k + k^2 + \ldots + k^n \leqslant nk^n$. Только эти строки могут быть в любом T_i. Значит, $T_m = T_{m-1}$ для некоторого $m \leqslant nk^n$. Таким образом, процедура, вычисляющая T_i для всех $i \geqslant 1$ до тех пор, пока не будут найдены два равных множества, гарантированно заканчивается, значит, это алгоритм. ■

Глава 3

Лексический анализ

Основная задача лексического анализа – разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов, или лексем, т.е. выделить эти слова из непрерывной последовательности символов. Все символы входной последовательности с этой точки зрения разделяются на символы, принадлежащие каким-либо лексемам, и символы, разделяющие лексемы (разделители). В некоторых случаях между лексемами может и не быть разделителей. С другой стороны, в некоторых языках лексемы могут содержать незначащие символы (например, символ пробела в Фортране). В Си разделительное значение символов-разделителей может блокироваться (“\” в конце строки внутри "...").

Обычно все лексемы делятся на классы. Примерами таких классов являются числа (целые, восьмеричные, шестнадцатиричные, действительные и т.д.), идентификаторы, строки. Отдельно выделяются ключевые слова и символы пунктуации (иногда их называют символы-ограничители). Как правило, ключевые слова – это некоторое конечное подмножество идентификаторов. В некоторых языках (например, ПЛ/1) смысл лексемы может зависеть от ее контекста и невозможно провести лексический анализ в отрыве от синтаксического.

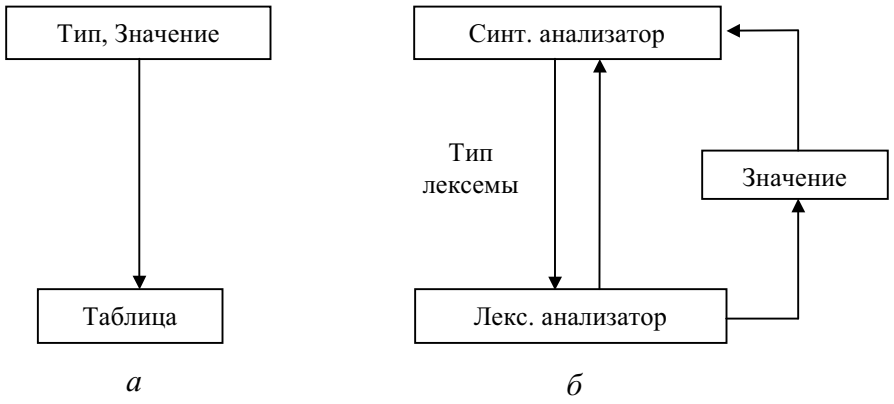
С точки зрения дальнейших фаз анализа лексический анализатор выдает информацию двух сортов: для синтаксического анализатора, работающего вслед за лексическим, существенна информация о последовательности классов лексем, ограничителей и ключевых слов, а для контекстного анализа, работающего вслед за синтаксическим, важна информация о конкретных значениях отдельных лексем (идентификаторов, чисел и т.д.).

Таким образом, общая схема работы лексического анализатора такова. Сначала выделяется отдельная лексема (возможно, используя символы-разделители). Ключевые слова распознаются либо явным выделением непосредственно из текста, либо сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых

слов.

Если выделенная лексема является ограничителем, то он (точнее, некоторый его признак) выдается как результат лексического анализа. Если выделенная лексема является ключевым словом, то выдается признак соответствующего ключевого слова. Если выделенная лексема является идентификатором – выдается признак идентификатора, а сам идентификатор сохраняется отдельно. Наконец, если выделенная лексема принадлежит какому-либо из других классов лексем (например, лексема представляет собой число, строку и т.д.), то выдается признак соответствующего класса, а значение лексемы сохраняется отдельно.

Лексический анализатор может быть как самостоятельной фазой трансляции, так и подпрограммой, работающей по принципу “дай лексему”. В первом случае (рис. 3.1, а) выходом анализатора является файл лексем, во втором (рис. 3.1, б) лексема выдается при каждом обращении к анализатору (при этом, как правило, признак класса лексемы возвращается как результат функции “лексический анализатор”, а значение лексемы передается через глобальную переменную). С точки зрения обработки значений лексем, анализатор может либо просто выдавать значение каждой лексемы, и в этом случае построение таблиц объектов (идентификаторов, строк, чисел и т.д.) переносится на более поздние фазы, либо он может самостоятельно строить таблицы объектов. В этом случае в качестве значения лексемы выдается указатель на вход в соответствующую таблицу.



а *б*

Рис. 3.1:

Работа лексического анализатора задается некоторым конечным автоматом. Однако, непосредственное описание конечного автомата неудобно с практической точки зрения. Поэтому для задания лексического анализатора, как правило, используется либо регулярное выражение, либо праволинейная грамматика. Все три формализма (конечных автоматов,

регулярных выражений и праволинейных грамматик) имеют одинаковую выразительную мощность. В частности, по регулярному выражению или праволинейной грамматике можно сконструировать конечный автомат, распознающий тот же язык.

3.1 Регулярные множества и выражения

Введем понятие *регулярного множества*, играющего важную роль в теории формальных языков.

Регулярное множество в алфавите T определяется рекурсивно следующим образом:

(1) $\varnothing$ **(пустое множество)** – регулярное множество в алфавите T;

(2) $\{e\}$ – регулярное множество в алфавите T (e – пустая цепочка);

(3) $\{a\}$ – регулярное множество в алфавите T для каждого $a \in T$;

(4) если P и Q – регулярные множества в алфавите T, то регулярными являются и множества

- **(а)** $P \cup Q$ **(объединение)**,
- **(б)** PQ **(конкатенация, т.е. множество** $\{pq | p \in P, q \in Q\}$**)**,
- **(в)** P^* **(итерация:** $P^* = \bigcup\limits_{n=0}^{\infty} P^n$**)**;

(5) ничто другое не является регулярным множеством в алфавите T.

Итак, множество в алфавите T регулярно тогда и только тогда, когда оно либо $\varnothing$, либо $\{e\}$, либо $\{a\}$ для некоторого $a \in T$, либо его можно получить из этих множеств применением конечного числа операций объединения, конкатенации и итерации.

Приведенное выше определение регулярного множества позволяет ввести следующую удобную форму его записи, называемую *регулярным выражением*.

Регулярное выражение в алфавите T и обозначаемое им регулярное множество в алфавите T определяются рекурсивно следующим образом:

(1) $\varnothing$ – регулярное выражение, обозначающее множество $\varnothing$;

(2) e – регулярное выражение, обозначающее множество $\{e\}$;

(3) a – регулярное выражение, обозначающее множество $\{a\}$;

(4) если p и q – регулярные выражения, обозначающие регулярные множества P и Q соответственно, то

- **(а)** $(p|q)$ – регулярное выражение, обозначающее регулярное множество $P \cup Q$,

(б) (pq) – регулярное выражение, обозначающее регулярное множество PQ,

(в) (p^*) – регулярное выражение, обозначающее регулярное множество P^*;

(5) ничто другое не является регулярным выражением в алфавите T.

Мы будем опускать лишние скобки в регулярных выражениях, договорившись о том, что операция итерации имеет наивысший приоритет, затем идет операции конкатенации, наконец, операция объединения имеет наименьший приоритет.

Кроме того, мы будем пользоваться записью p^+ для обозначения pp^*. Таким образом, запись $(a|((ba)(a^*)))$ эквивалентна $a|ba^+$.

Наконец, мы будем использовать запись $L(r)$ для регулярного множества, обозначаемого регулярным выражением r.

Пример 3.1. Несколько примеров регулярных выражений и обозначаемых ими регулярных множеств:

а) $a(e|a)|b$ – обозначает множество $\{a, b, aa\}$;

б) $a(a|b)^*$ – обозначает множество всевозможных цепочек, состоящих из a и b, начинающихся с a;

в) $(a|b)^*(a|b)(a|b)^*$ – обозначает множество всех непустых цепочек, состоящих из a и b, т.е. множество $\{a, b\}^+$;

г) $((0|1)(0|1)(0|1))^*$ – обозначает множество всех цепочек, состоящих из нулей и единиц, длины которых делятся на 3.

Ясно, что для каждого регулярного множества можно найти регулярное выражение, обозначающее это множество, и наоборот. Более того, для каждого регулярного множества существует бесконечно много обозначающих его регулярных выражений.

Будем говорить, что регулярные выражения *равны* или *эквивалентны* (=), если они обозначают одно и то же регулярное множество.

Существует ряд алгебраических законов, позволяющих осуществлять эквивалентное преобразование регулярных выражений.

Лемма. *Пусть p, q и r – регулярные выражения. Тогда справедливы следующие соотношения:*

(1)	$p\|q = q\|p$;	(7)	$pe = ep = p$;
(2)	$\varnothing^* = e$;	(8)	$\varnothing p = p\varnothing = \varnothing$;
(3)	$p\|(q\|r) = (p\|q)\|r$;	(9)	$p^* = p\|p^*$;
(4)	$p(qr) = (pq)r$;	(10)	$(p^*)^* = p^*$;
(5)	$p(q\|r) = pq\|pr$;	(11)	$p\|p = p$;
(6)	$(p\|q)r = pr\|qr$;	(12)	$p\|\varnothing = p$.

Следствие. *Для любого регулярного выражения существует эквивалентное регулярное выражение, которое либо есть $\varnothing$, либо не содержит в своей записи $\varnothing$.*

В дальнейшем будем рассматривать только регулярные выражения, не содержащие в своей записи $\varnothing$.

При практическом описании лексических структур бывает полезно сопоставлять регулярным выражениям некоторые имена, и ссылаться на них по этим именам. Для определения таких имен мы будем использовать запись вида

$$d_1 = r_1$$
$$d_2 = r_2$$
$$\ldots$$
$$d_n = r_n$$

где d_i – различные имена, а каждое r_i – регулярное выражение над символами $T \cup \{d_1, d_2, \ldots, d_{i-1}\}$, т.е. символами основного алфавита и ранее определенными символами (именами). Таким образом, для любого r_i можно построить регулярное выражение над T, повторно заменяя имена регулярных выражений на обозначаемые ими регулярные выражения.

Пример 3.2. Использование имен для регулярных выражений.

а) Регулярное выражение для множества идентификаторов.

$$Letter = a|b|c|\ldots|x|y|z$$
$$Digit = 0|1|\ldots|9$$
$$Identifier = Letter(Letter|Digit)^*$$

б) Регулярное выражение для множества чисел в десятичной записи.

$$Digit = 0|1|\ldots|9$$
$$Integer = Digit^+$$
$$Fraction = .Integer|e$$
$$Exponent = (E(+|-|e)Integer)|e$$
$$Number = Integer\ Fraction\ Exponent$$

3.2 Конечные автоматы

Регулярные выражения, введенные ранее, служат для *описания* регулярных множеств. Для *распознавания* регулярных множеств служат *конечные автоматы*.

Недетерминированный конечный автомат (НКА) – это пятерка $M = (Q, T, D, q_0, F)$, где

(1) Q – конечное множество *состояний*;

(2) T – конечное множество допустимых *входных символов* (входной алфавит);

(3) D – *функция переходов* (отображающая множество $Q \times (T \cup \{e\})$ во множество подмножеств множества Q), определяющая поведение управляющего устройства;

(4) $q_0 \in Q$ – *начальное состояние* управляющего устройства;

(5) $F \subseteq Q$ – множество *заключительных состояний*.

Работа конечного автомата представляет собой некоторую последовательность шагов, или тактов. Такт определяется текущим состоянием управляющего устройства и входным символом, обозреваемым в данный момент входной головкой. Сам шаг состоит из изменения состояния и, возможно, сдвига входной головки на одну ячейку вправо (рис. 3.2).

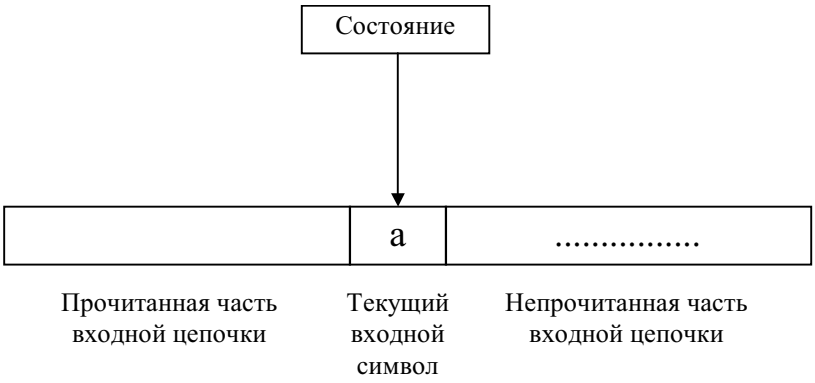


Рис. 3.2:

Недетерминизм автомата заключается в том, что, во-первых, находясь в некотором состоянии и обозревая текущий символ, автомат может перейти в одно из, вообще говоря, *нескольких* возможных состояний, и во-вторых, автомат может делать переходы по e.

Пусть $M = (Q, T, D, q_0, F)$ – НКА. *Конфигурацией* автомата M называется пара $(q, w) \in Q \times T^*$, где q – текущее состояние управляющего устройства, а w – цепочка символов на входной ленте, состоящая из символа под головкой и всех символов справа от него. Конфигурация (q_0, w) называется *начальной*, а конфигурация (q, e), где $q \in F$ – *заключительной* (или *допускающей*).

Пусть $M = (Q, T, D, q_0, F)$ – НКА. *Тактом* автомата M называется бинарное отношение $\vdash$, определенное на конфигурациях M следующим образом: если $p \in D(q, a)$, где $a \in T \cup \{e\}$, то $(q, aw) \vdash (p, w)$ для всех $w \in T^*$.

Будем обозначать символом $\vdash^+$ ($\vdash^*$) транзитивное (рефлексивно-транзитивное) замыкание отношения $\vdash$.

Говорят, что автомат M *допускает* цепочку w, если $(q_0, w) \vdash^* (q, e)$ для некоторого $q \in F$. Языком, *допускаемым* (*распознаваемым*, *определяемым*) автоматом M, (обозначается $L(M)$), называется множество входных цепочек, допускаемых автоматом M. Т.е.

$$L(M) = \{w | w \in T^* \text{ и } (q_0, w) \vdash^* (q, e) \text{ для некоторого } q \in F\}.$$

Важным частным случаем недетерминированного конечного автомата является *детерминированный конечный автомат*, который на каждом такте работы имеет возможность перейти не более чем в одно состояние и не может делать переходы по e.

Пусть $M = (Q, T, D, q_0, F)$ – НКА. Будем называть M *детерминированным конечным автоматом* (ДКА), если выполнены следующие два условия:

(1) $D(q, e) = \varnothing$ для любого $q \in Q$, и

(2) $D(q, a)$ содержит не более одного элемента для любых $q \in Q$ и $a \in T$.

Так как функция переходов ДКА содержит не более одного элемента для любой пары аргументов, для ДКА мы будем пользоваться записью $D(q, a) = p$ вместо $D(q, a) = \{p\}$.

Конечный автомат может быть изображен графически в виде *диаграммы*, представляющей собой ориентированный граф, в котором каждому состоянию соответствует вершина, а дуга, помеченная символом $a \in T \cup \{e\}$, соединяет две вершины p и q, если $p \in D(q, a)$. На диаграмме выделяются начальное и заключительные состояния (в примерах ниже, соответственно, входящей стрелкой и двойным контуром).

Пример 3.3. Пусть $L = L(r)$, где $r = (a|b)^*a(a|b)(a|b)$.

а) Недетерминированный конечный автомат M, допускающий язык L:

$$M = \{\{1, 2, 3, 4\}, \{a, b\}, D, 1, \{4\}\},$$

где функция переходов D определяется так:

$$D(1, a) = \{1, 2\}, \quad D(3, a) = \{4\},$$
$$D(2, a) = \{3\}, \quad D(3, b) = \{4\},$$
$$D(2, b) = \{3\}.$$

Диаграмма автомата приведена на рис. 3.3, а.

б) Детерминированный конечный автомат M, допускающий язык L:

$$M = \{\{1, 2, 3, 4, 5, 6, 7, 8\}, \{a, b\}, D, 1, \{3, 5, 6, 8\}\},$$

где функция переходов D определяется так:

$$D(1, a) = 2, \quad D(5, a) = 8,$$
$$D(1, b) = 1, \quad D(5, b) = 6,$$
$$D(2, a) = 4, \quad D(6, a) = 2,$$
$$D(2, b) = 7, \quad D(6, b) = 1,$$
$$D(3, a) = 3, \quad D(7, a) = 8,$$
$$D(3, b) = 5, \quad D(7, b) = 6,$$
$$D(4, a) = 3, \quad D(8, a) = 4,$$
$$D(4, b) = 5, \quad D(8, b) = 7.$$

Диаграмма автомата приведена на рис. 3.3, б.

Пример 3.4. Диаграмма ДКА, допускающего множество чисел в десятичной записи, приведена на рис. 3.4.

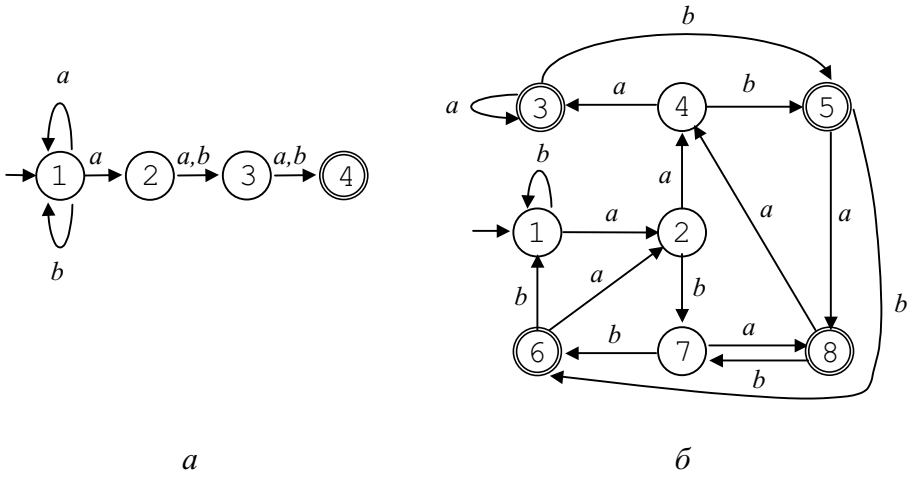


Рис. 3.3:

Пример 3.5. Анализ цепочек.

а) При анализе цепочки $w = ababa$ автомат из примера 3.3, а, может сделать следующую последовательность тактов:
$(1, ababa) \vdash (1, baba) \vdash (1, aba) \vdash (2, ba) \vdash (3, a) \vdash (4, e)$.
Состояние 4 является заключительным, следовательно, цепочка w допускается этим автоматом.

б) При анализе цепочки $w = ababab$ автомат из примера 3.3, б, должен сделать следующую последовательность тактов:
$(1, ababab) \vdash (2, babab) \vdash (7, abab) \vdash (8, bab) \vdash (7, ab) \vdash (8, b) \vdash (7, e)$.
Так как состояние 7 не является заключительным, цепочка w не допускается этим автоматом.

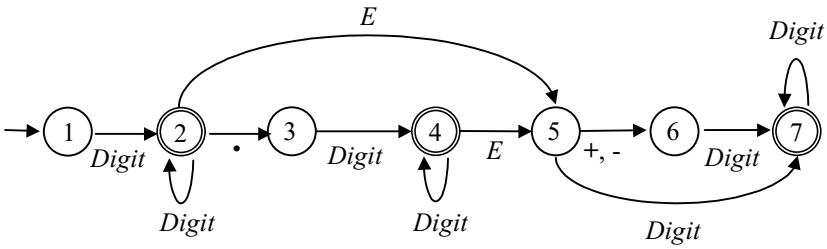


Рис. 3.4:

3.3 Алгоритмы построения конечных автоматов

3.3.1 Построение недетерминированного конечного автомата по регулярному выражению

Рассмотрим алгоритм построения по регулярному выражению недетерминированного конечного автомата, допускающего тот же язык.

Алгоритм 3.1. Построение недетерминированного конечного автомата по регулярному выражению.

Вход. Регулярное выражение r в алфавите T.

Выход. НКА M, такой что $L(M) = L(r)$.

Метод. Автомат для выражения строится композицией из автоматов, соответствующих подвыражениям. На каждом шаге построения строящийся автомат имеет в точности одно заключительное состояние, в начальное состояние нет переходов из других состояний и нет переходов из заключительного состояния в другие.

1. Для выражения e строится автомат

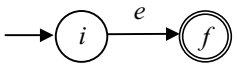


Рис. 3.5:

2. Для выражения a ($a \in T$) строится автомат

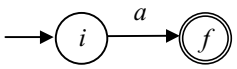


Рис. 3.6:

3. Пусть $M(s)$ и $M(t)$ – НКА для регулярных выражений s и t соответственно.

   а) Для выражения $s|t$ автомат $M(s|t)$ строится как показано на рис. 3.7. Здесь i – новое начальное состояние и f – новое заключительное состояние. Заметим, что имеет место переход по e из i в начальные состояния $M(s)$ и $M(t)$ и переход по e из заключительных состояний $M(s)$ и $M(t)$ в f. Начальное и заключительное состояния автоматов $M(s)$ и $M(t)$ не являются таковыми для автомата $M(s|t)$.

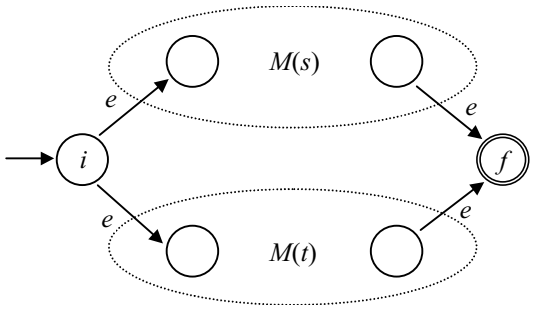


Рис. 3.7:

б) Для выражения st автомат $M(st)$ строится следующим образом:

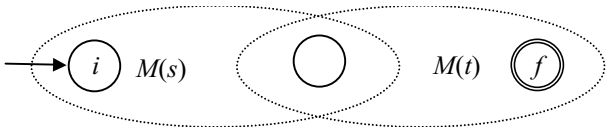


Рис. 3.8:

Начальное состояние $M(s)$ становится начальным для нового автомата, а заключительное состояние $M(t)$ становится заключительным для нового автомата. Начальное состояние $M(t)$ и заключительное состояние $M(s)$ сливаются, т.е. все переходы из начального состояния $M(t)$ становятся переходами из заключительного состояния $M(s)$. В новом автомате это объединенное состояние не является ни начальным, ни заключительным.

в) Для выражения s^* автомат $M(s^*)$ строится следующим образом:

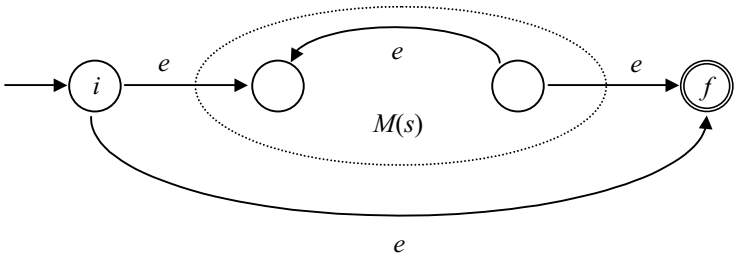


Рис. 3.9:

Здесь i – новое начальное состояние, а f – новое заключительное состояние.

3.3.2 Построение детерминированного конечного автомата по недетерминированному

Рассмотрим алгоритм построения по недетерминированному конечному автомату детерминированного конечного автомата, допускающего тот же язык.

Алгоритм 3.2. Построение детерминированного конечного автомата по недетерминированному.

Вход. НКА $M = (Q, T, D, q_0, F)$.

Выход. ДКА $M' = (Q', T, D', q'_0, F')$, такой что $L(M) = L(M')$.

Метод. Каждое состояние результирующего ДКА – это некоторое множество состояний исходного НКА.

В алгоритме будут использоваться следующие функции:

$e\text{-}closure(R)$ ($R \subseteq Q$) – множество состояний НКА, достижимых из состояний, входящих в R, посредством только переходов по e, т.е. множество

$$S = \bigcup_{q \in R} \{p | (q, e) \vdash^* (p, e)\}$$

$move(R, a)$ ($R \subseteq Q$) – множество состояний НКА, в которые есть переход на входе a для состояний из R, т.е. множество

$$S = \bigcup_{q \in R} \{p | p \in D(q, a)\}$$

Вначале Q' и D' пусты. Выполнить шаги 1-4:

(1) Определить $q'_0 = e\text{-}closure(\{q_0\})$.

(2) Добавить q'_0 в Q' как непомеченное состояние.

(3) Выполнить следующую процедуру:

while (в Q' есть непомеченное состояние R){
  пометить R;
  for (каждого входного символа $a \in T$){
    $S = e\text{-}closure(move(R, a))$;
    if ($S \neq \varnothing$){
      if ($S \notin Q'$)
        добавить S в Q' как непомеченное состояние;
      определить $D'(R, a) = S$;

```
        }
      }
    }
```

(4) Определить $F' = \{S | S \in Q', S \cap F \neq \varnothing\}$.

Пример 3.6. Результат применения алгоритма 3.2 приведен на рис. 3.10.

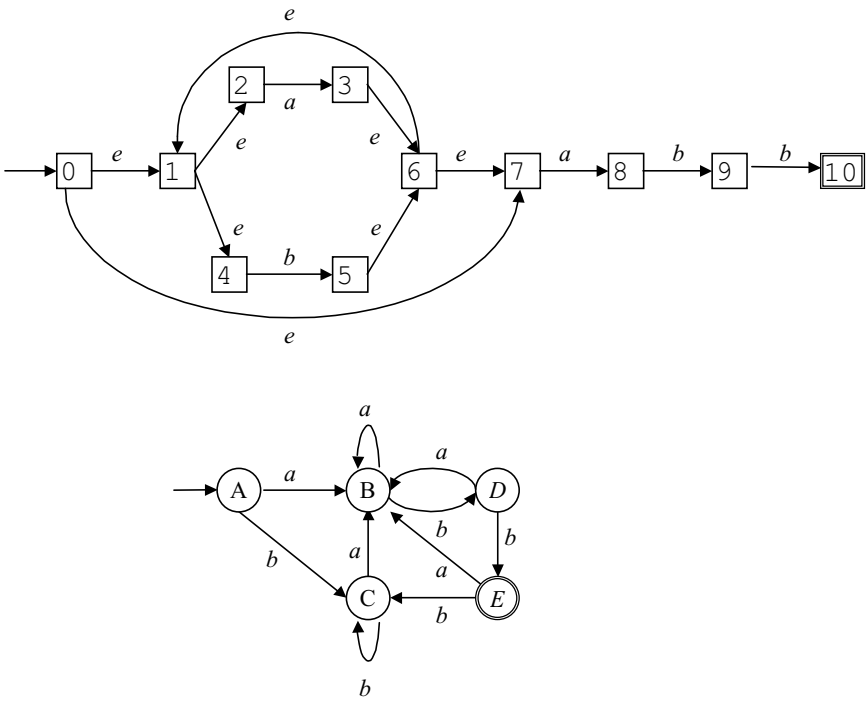


Рис. 3.10:

3.3.3 Построение детерминированного конечного автомата по регулярному выражению

Приведем теперь алгоритм построения по регулярному выражению детерминированного конечного автомата, допускающего тот же язык [10].

Пусть дано регулярное выражение r в алфавите T. К регулярному выражению r добавим маркер конца: $(r)\#$. Такое регулярное выражение будем называть *пополненным*. В процессе своей работы алгоритм будет использовать пополненное регулярное выражение.

Алгоритм будет оперировать с синтаксическим деревом для пополненного регулярного выражения $(r)\#$, каждый лист которого помечен символом $a \in T \cup \{e, \#\}$, а каждая внутренняя вершина помечена знаком одной из операций: $\cdot$ (конкатенация), $|$ (объединение), * (итерация).

Каждому листу дерева (кроме e-листьев) припишем уникальный номер, называемый *позицией*, и будем использовать его, с одной стороны, для ссылки на лист в дереве, и, с другой стороны, для ссылки на символ, соответствующий этому листу. Заметим, что если некоторый символ используется в регулярном выражении несколько раз, он имеет несколько позиций.

Теперь, обходя дерево T снизу-вверх слева-направо, вычислим четыре функции: $nullable$, $firstpos$, $lastpos$ и $followpos$. Функции $nullable$, $firstpos$ и $lastpos$ определены на узлах дерева, а $followpos$ – на множестве позиций. Значением всех функций, кроме $nullable$, является множество позиций. Функция $followpos$ вычисляется через три остальные функции.

Функция $firstpos(n)$ для каждого узла n синтаксического дерева регулярного выражения дает множество позиций, которые соответствуют первым символам в подцепочках, генерируемых подвыражением с вершиной в n. Аналогично, $lastpos(n)$ дает множество позиций, которым соответствуют последние символы в подцепочках, генерируемых подвыражениями с вершиной n. Для узла n, поддеревья которого (т.е. деревья, у которых узел n является корнем) могут породить пустое слово, определим $nullable(n) = true$, а для остальных узлов $nullable(n) = false$.

Таблица для вычисления функций $nullable$, $firstpos$ и $lastpos$ приведена на рис. 3.11.

узел n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
лист e	$true$	$\varnothing$	$\varnothing$
лист i (не e)	$false$	$\{i\}$	$\{i\}$
$\vert$ /\ u v	$nullable(u)$ or $nullable(v)$	$firstpos(u) \cup firstpos(v)$	$lastpos(u) \cup lastpos(v)$
$\cdot$ /\ u v	$nullable(u)$ and $nullable(v)$	if $nullable(u)$ then $firstpos(u) \cup firstpos(v)$ else $firstpos(u)$	if $nullable(v)$ then $lastpos(u) \cup lastpos(v)$ else $lastpos(v)$
* | v	$true$	$firstpos(v)$	$lastpos(v)$

Рис. 3.11:

Пример 3.7. Синтаксическое дерево для пополненного регулярного выражения $(a|b)^*abb\#$ с результатом вычисления функций $firstpos$ и $lastpos$ приведено

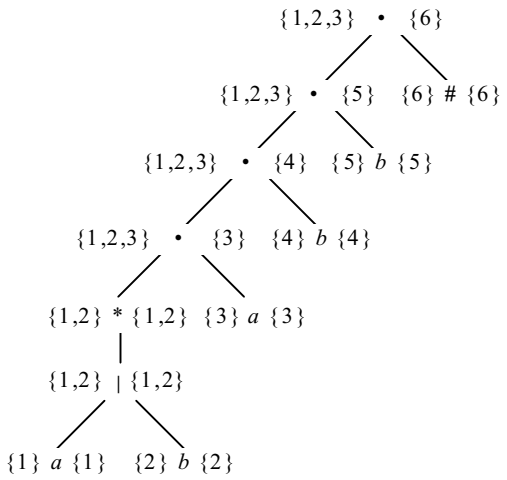


Рис. 3.12:

позиция	$followpos$
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	$\varnothing$

Рис. 3.13:

на рис. 3.12. Слева от каждого узла расположено значение $firstpos$, справа от узла – значение $lastpos$. Заметим, что эти функции могут быть вычислены за один обход дерева.

Если i – позиция, то $followpos(i)$ есть множество позиций j таких, что существует некоторая строка $\ldots cd \ldots$, входящая в язык, описываемый регулярным выражением, такая, что позиция i соответствует этому вхождению c, а позиция j – вхождению d.

Функция $followpos$ может быть вычислена также за один обход дерева снизу-вверх по следующим двум правилам.

1. Пусть n – внутренний узел с операцией $\cdot$ (конкатенация), u и v – его потомки. Тогда для каждой позиции i, входящей в $lastpos(u)$, добавляем к множеству значений $followpos(i)$ множество $firstpos(v)$.

2. Пусть n – внутренний узел с операцией * (итерация), u – его потомок. Тогда для каждой позиции i, входящей в $lastpos(u)$, добавляем к множеству значений $followpos(i)$ множество $firstpos(u)$.

Пример 3.8. Результат вычисления функции $followpos$ для регулярного выражения из предыдущего примера приведен на рис. 3.13.

Алгоритм 3.3. Прямое построение ДКА по регулярному выражению.
Вход. Регулярное выражение r в алфавите T.
Выход. ДКА $M = (Q, T, D, q_0, F)$, такой что $L(M) = L(r)$.
Метод. Состояния ДКА соответствуют множествам позиций.
Вначале Q и D пусты. Выполнить шаги 1-6:

(1) Построить синтаксическое дерево для пополненного регулярного выражения $(r)\#$.

(2) Обходя синтаксическое дерево, вычислить значения функций $nullable$, $firstpos$, $lastpos$ и $followpos$.

(3) Определить $q_0 = firstpos(root)$, где $root$ – корень синтаксического дерева.

(4) Добавить q_0 в Q как непомеченное состояние.

(5) Выполнить следующую процедуру:

while (в Q есть непомеченное состояние R){
  пометить R;
  for (каждого входного символа $a \in T$, такого, что
      в R имеется позиция, которой соответствует a){
    пусть символ a в R соответствует позициям
    $p_1, \ldots, p_n$, и пусть $S = \bigcup_{1 \le i \le n} followpos(p_i)$;
    if ($S \neq \varnothing$){
      if ($S \notin Q$)
        добавить S в Q как непомеченное состояние;
      определить $D(R, a) = S$;
    }
  }
}

(6) Определить F как множество всех состояний из Q, содержащих позиции, связанные с символом $\#$.

Пример 3.9. Результат применения алгоритма 3.3 для регулярного выражения $(a|b)^*abb$ приведен на рис. 3.14.

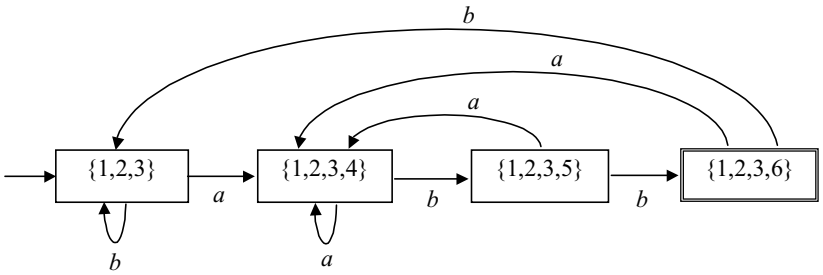


Рис. 3.14:

3.3.4 Построение детерминированного конечного автомата с минимальным числом состояний

Рассмотрим теперь алгоритм построения ДКА с минимальным числом состояний, эквивалентного данному ДКА [10].

Пусть $M = (Q, T, D, q_0, F)$ – ДКА. Будем называть M *всюду определенным*, если $D(q, a) \neq \varnothing$ для всех $q \in Q$ и $a \in T$.

Лемма. *Пусть* $M = (Q, T, D, q_0, F)$ – ДКА, *не являющийся всюду определенным. Существует всюду определенный* ДКА M', *такой что* $L(M) = L(M')$.

Доказательство. Рассмотрим автомат $M' = (Q \cup \{q'\}, T, D', q_0, F)$, где $q' \notin Q$ – некоторое новое состояние, а функция D' определяется следующим образом:

(1) Для всех $q \in Q$ и $a \in T$, таких что $D(q, a) \neq \varnothing$, определить $D'(q, a) = D(q, a)$.

(2) Для всех $q \in Q$ и $a \in T$, таких что $D(q, a) = \varnothing$, определить $D'(q, a) = q'$.

(3) Для всех $a \in T$ определить $D'(q', a) = q'$.

Легко показать, что автомат M' допускает тот же язык, что и M. ■

Приведенный ниже алгоритм получает на входе всюду определенный автомат. Если автомат не является всюду определенным, его можно сделать таковым на основании только что приведенной леммы.

Алгоритм 3.4. Построение ДКА с минимальным числом состояний.

Вход. Всюду определенный ДКА $M = (Q, T, D, q_0, F)$.

Выход. ДКА $M' = (Q', T, D', q_0', F')$, такой что $L(M) = L(M')$ и M' содержит наименьшее возможное число состояний.

Метод. Выполнить шаги 1-5:

(1) Построить начальное разбиение Π множества состояний из двух групп: заключительные состояния Q и остальные $Q-F$, т.е. $\Pi = \{F, Q-F\}$.

(2) Применить к Π следующую процедуру и получить новое разбиение Π_{new}:

for (каждой группы G в Π){

разбить G на подгруппы так, чтобы

состояния s и t из G оказались

в одной подгруппе тогда и только тогда,

когда для каждого входного символа a

состояния s и t имеют переходы по a

в состояния из одной и той же группы в Π;

заменить G в Π_{new} на множество всех

полученных подгрупп;

}

(3) Если $\Pi_{\text{new}} = \Pi$, полагаем $\Pi_{\text{res}} = \Pi$ и переходим к шагу 4, иначе повторяем шаг 2 с $\Pi := \Pi_{\text{new}}$.

(4) Пусть $\Pi_{\text{res}} = \{G_1, \ldots, G_n\}$. Определим:

$Q' = \{G_1, \ldots, G_n\}$;

$q'_0 = G$, где группа $G \in Q'$ такова, что $q_0 \in G$;

$F' = \{G | G \in Q' \text{ и } G \cap F \neq \varnothing\}$;

$D'(p', a) = q'$, если $D(p, a) = q$, где $p \in p'$ и $q \in q'$.

Таким образом, каждая группа в Π_{res} становится состоянием нового автомата M'. Если группа содержит начальное состояние автомата M, эта группа становится начальным состоянием автомата M'. Если группа содержит заключительное состояние M, она становится заключительным состоянием M'. Отметим, что каждая группа Π_{res} либо состоит только из состояний из F, либо не имеет состояний из F. Переходы определяются очевидным образом.

(5) Если M' имеет "мертвое" состояние, т.е. состояние, которое не является допускающим и из которого нет путей в допускающие, удалить его и связанные с ним переходы из M'. Удалить из M' также все состояния, недостижимые из начального.

Пример 3.10. Результат применения алгоритма 3.4 приведен на рис. 3.15.

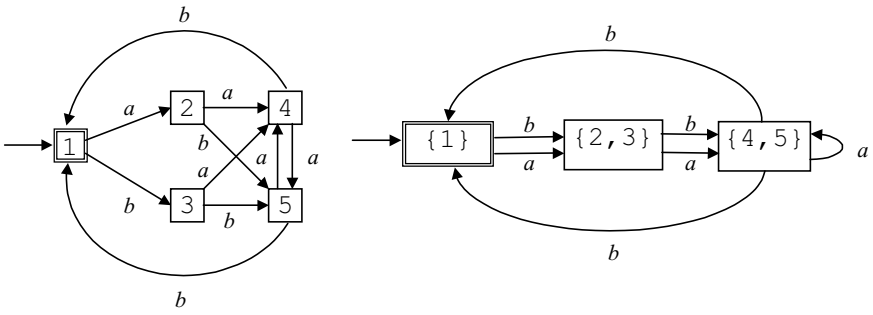


Рис. 3.15:

3.4 Регулярные множества и их представления

В разделе 3.3.3 приведен алгоритм построения детерминированного конечного автомата по регулярному выражению. Рассмотрим теперь как по описанию конечного автомата построить регулярное множество, совпадающее с языком, допускаемым конечным автоматом.

Теорема 3.1. *Язык, допускаемый детерминированным конечным автоматом, является регулярным множеством.*

Доказательство. Пусть L – язык допускаемый детерминированным конечным автоматом $M = (\{q_1, \dots, q_n\}, T, D, q_1, F)$. Введем D^e – расширенную функцию переходов автомата M: $D^e(q, w) = p$, где $w \in T^*$, тогда и только тогда, когда $(q, w) \vdash^* (p, e)$.

Обозначим R_{ij}^k множество всех слов x таких, что $D^e(q_i, x) = q_j$ и если $D^e(q_i, y) = q_s$ для любой цепочки y – префикса x, отличного от x и e, то $s \leqslant k$.

Иными словами, R_{ij}^k есть множество всех слов, которые переводят конечный автомат из состояния q_i в состояние q_j, не проходя ни через какое состояние q_s для $s > k$. Однако, i и j могут быть больше k.

R_{ij}^k может быть определено рекурсивно следующим образом:

$$R_{ij}^0 = \{a | a \in T, D(q_i, a) = q_j\},$$

$$R_{ij}^k = R_{ij}^{k-1} \bigcup R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1}, \text{ где } 1 \leqslant k \leqslant n.$$

Таким образом, определение R_{ij}^k означает, что для входной цепочки w, переводящей M из q_i в q_j без перехода через состояния с номерами, большими k, справедливо ровно одно из следующих двух утверждений:

1. Цепочка w принадлежит R_{ij}^{k-1}, т.е. при анализе цепочки w автомат никогда не достигает состояний с номерами, большими или равными k.

2. Цепочка w может быть представлена в виде $w = w_1w_2w_3$, где $w_1 \in R_{ik}^{k-1}$ (подцепочка w_1 переводит M сначала в q_k), $w_2 \in (R_{kk}^{k-1})^*$ (подцепочка w_2 переводит M из q_k обратно в q_k, не проходя через состояния с номерами, большими или равными k), и $w_3 \in R_{kj}^{k-1}$ (подцепочка w_3 переводит M из состояния q_k в q_j) – рис. 3.16.

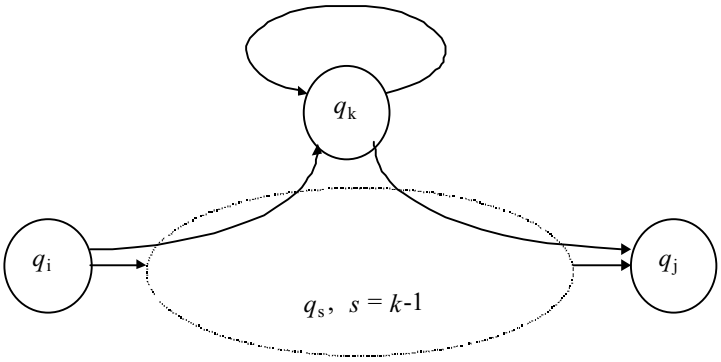


Рис. 3.16:

Тогда $L = \bigcup\limits_{q_j \in F} R_{1j}^n$. Индукцией по k можно показать, что это множество является регулярным. ■

Таким образом, для всякого регулярного множества имеется конечный автомат, допускающий в точности это регулярное множество, и наоборот – язык, допускаемый конечным автоматом есть регулярное множество.

Рассмотрим теперь соотношение между языками, порождаемыми праволинейными грамматиками и допускаемыми конечными автоматами.

Праволинейная грамматика $G = (N, T, P, S)$ называется *регулярной*, если

(1) каждое ее правило, кроме $S \to e$, имеет вид либо $A \to aB$, либо $A \to a$, где $A, B \in N$, $a \in T$;

(2) в том случае, когда $S \to e \in P$, начальный символ S не встречается в правых частях правил.

Лемма. *Пусть G – праволинейная грамматика. Существует регулярная грамматика G' такая, что $L(G) = L(G')$.*

Доказательство. Предоставляется читателю в качестве упражнения. ■

Теорема 3.2. *Пусть $G = (N, T, P, S)$ – праволинейная грамматика. Тогда существует конечный автомат $M = (Q, T, D, q_0, F)$ для которого $L(M) = L(G)$.*

Доказательство. На основании приведенной выше леммы, без ограничения общности можно считать, что G – регулярная грамматика.

Построим недетерминированный конечный автомат M следующим образом:

1. состояниями M будут нетерминалы G плюс новое состояние R, не принадлежащее N. Так что $Q = N \cup \{R\}$;
2. в качестве начального состояния M примем S, т.е. $q_0 = S$;
3. если P содержит правило $S \to e$, то $F = \{S, R\}$, иначе $F = \{R\}$. Напомним, что S не встречается в правых частях правил, если $S \to e \in P$;
4. состояние $R \in D(A, a)$, если $A \to a \in P$. Кроме того, $D(A, a)$ содержит все B такие, что $A \to aB \in P$. $D(R, a) = \varnothing$ для каждого $a \in T$.

M, читая вход w, моделирует вывод w в грамматике G. Покажем, что $L(M) = L(G)$. Пусть $w = a_1 a_2 \ldots a_n \in L(G)$, $n \geqslant 1$. Тогда

$$S \Rightarrow a_1 A_1 \Rightarrow \ \ldots \ \Rightarrow a_1 a_2 \ldots a_{n-1} A_{n-1} \Rightarrow a_1 a_2 \ldots a_{n-1} a_n$$

для некоторой последовательности нетерминалов $A_1, A_2, \ldots, A_{n-1}$. По определению, $D(S, a_1)$ содержит A_1, $D(A_1, a_2)$ содержит A_2, и т.д., $D(A_{n-1}, a_n)$ содержит R. Так что $w \in L(M)$, поскольку $D^e(S, w)$ содержит R, а $R \in F$. Если $e \in L(G)$, то $S \in F$, так что $e \in L(M)$.

Аналогично, если $w = a_1 a_2 \ldots a_n \in L(M)$, $n \geqslant 1$, то существует последовательность состояний $S, A_1, A_2, \ldots, A_{n-1}, R$ такая, что $D(S, a_1)$ содержит A_1, $D(A_1, a_2)$ содержит A_2, и т.д. Поэтому

$$S \Rightarrow a_1 A_1 \Rightarrow a_1 a_2 A_2 \Rightarrow \ \ldots \ \Rightarrow a_1 a_2 \ldots a_{n-1} A_{n-1} \Rightarrow a_1 a_2 \ldots a_{n-1} a_n$$

– вывод в G и $x \in L(G)$. Если $e \in L(M)$, то $S \in F$, так что $S \to e \in P$ и $e \in L(G)$. ■

Теорема 3.3. *Для каждого конечного автомата $M = (Q, T, D, q_0, F)$ существует праволинейная грамматика $G = (N, T, P, S)$ такая, что $L(G) = L(M)$.*

Доказательство. Без потери общности можно считать, что автомат M – детерминированный. Определим грамматику G следующим образом:

1. нетерминалами грамматики G будут состояния автомата M. Так что $N = Q$;
2. в качестве начального символа грамматики G примем q_0, т.е. $S = q_0$;
3. $A \to aB \in P$, если $D(A, a) = B$;

4. $A \to a \in P$, если $D(A, a) = B$ и $B \in F$;

5. $S \to e \in P$, если $q_0 \in F$.

Доказательство того, что $S \Rightarrow^* w$ тогда и только тогда, когда $D^e(q_0, w) \in F$, аналогично доказательству теоремы 3.2. ■

3.5 Программирование лексического анализа

Как уже отмечалось ранее, лексический анализатор (ЛА) может быть оформлен как подпрограмма. При обращении к ЛА, вырабатываются как минимум два результата: тип выбранной лексемы и ее значение (если оно есть).

Если ЛА сам формирует таблицы объектов, он выдает тип лексемы и указатель на соответствующий вход в таблице объектов. Если же ЛА не работает с таблицами объектов, он выдает тип лексемы, а ее значение передается, например, через некоторую глобальную переменную. Помимо значения лексемы, эта глобальная переменная может содержать некоторую дополнительную информацию: номер текущей строки, номер символа в строке и т.д. Эта информация может использоваться в различных целях, например, для диагностики.

В основе ЛА лежит диаграмма переходов соответствующего конечного автомата. Отдельная проблема здесь – анализ ключевых слов. Как правило, ключевые слова – это выделенные идентификаторы. Поэтому возможны два основных способа распознавания ключевых слов: либо очередная лексема сначала диагностируется на совпадение с каким-либо ключевым словом и в случае неуспеха делается попытка выделить лексему из какого-либо класса, либо, наоборот, после выборки лексемы идентификатора происходит обращение к таблице ключевых слов на предмет сравнения. Подробнее о механизмах поиска в таблицах будет сказано ниже (гл. 7), здесь отметим только, что поиск ключевых слов может вестись либо в основной таблице имен и в этом случае в нее до начала работы ЛА загружаются ключевые слова, либо в отдельной таблице. При первом способе все ключевые слова непосредственно встраиваются в конечный автомат лексического анализатора, во втором конечный автомат содержит только разбор идентификаторов.

В некоторых языках (например, ПЛ/1 или Фортран) ключевые слова могут использоваться в качестве обычных идентификаторов. В этом случае работа ЛА не может идти независимо от работы синтаксического анализатора. В Фортране возможны, например, следующие строки:

```
DO 10 I=1,25

DO 10 I=1.25
```

В первом случае строка – это заголовок цикла DO, во втором – оператор присваивания. Поэтому, прежде чем можно будет выделить лексему, лексический анализатор должен заглянуть довольно далеко.

Еще сложнее дело в ПЛ/1. Здесь возможны такие операторы:

```
IF ELSE THEN ELSE = THEN; ELSE THEN = ELSE;
```

или

```
DECLARE (A1, A2, ... , AN)
```

и только в зависимости от того, что стоит после ")", можно определить, является ли DECLARE ключевым словом или идентификатором. Длина такой строки может быть сколь угодно большой и уже невозможно отделить фазу синтаксического анализа от фазы лексического анализа.

Рассмотрим несколько подробнее вопросы программирования ЛА. Основная операция лексического анализатора, на которую уходит большая часть времени его работы – это взятие очередного символа и проверка на принадлежность его некоторому диапазону. Например, основной цикл при выборке числа в простейшем случае может выглядеть следующим образом:

```
while (Insym<='9' && Insym>='0')
{ ... }
```

Программу можно значительно улучшить следующим образом [4]. Пусть LETTER, DIGIT, BLANK – элементы перечислимого типа. Введем массив map, входами которого будут символы, значениями – типы символов. Инициализируем массив map следующим образом:

```
map['a']=LETTER;
   ........
map['z']=LETTER;
map['0']=DIGIT;
   ........
map['9']=DIGIT;
map[' ']=BLANK;
   ........
```

Тогда приведенный выше цикл примет следующую форму:

```
while (map[Insym]==DIGIT)
{ ... }
```

Выделение ключевых слов может осуществляться после выделения идентификаторов. ЛА работает быстрее, если ключевые слова выделяются непосредственно.

Для этого строится конечный автомат, описывающий множество ключевых слов. На рис. 3.17 приведен фрагмент такого автомата. Рассмотрим пример программирования этого конечного автомата на языке Си, приведенный в [14]:

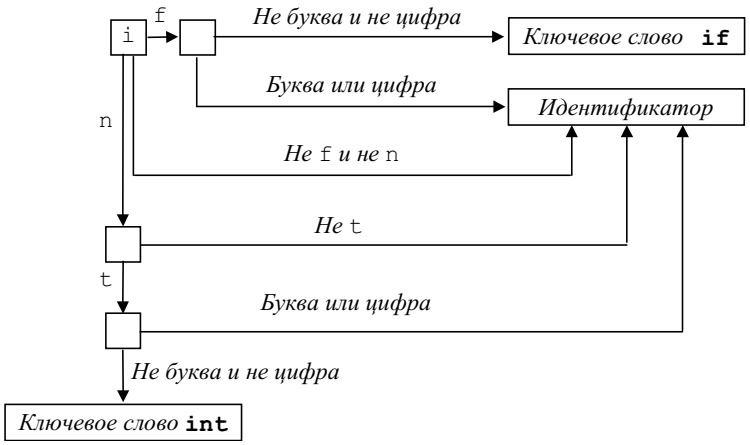


Рис. 3.17:

```
.....
case 'i':
 if (cp[0]=='f' &&!(map[cp] & (DIGIT | LETTER)))
 {cp++; return IF;}
 if (cp[0]=='n' &&  cp=='t'
  &&!(map[cp] & (DIGIT | LETTER)))
 {cp+=2;  return INT;}
 { обработка идентификатора }
.....
```

Здесь cp – указатель текущего символа. В массиве map классы символов кодируются битами.

Поскольку ЛА анализирует каждый символ входного потока, его скорость существенно зависит от скорости выборки очередного символа входного потока. В свою очередь, эта скорость во многом определяется схемой буферизации. Рассмотрим возможные эффективные схемы буферизации.

Будем использовать буфер, состоящий из двух одинаковых частей длины N (рис. 3.18, а), где N – размер блока обмена (например, 1024, 2048 и т.д.).

Чтобы не читать каждый символ отдельно, в каждую из половин буфера поочередно одной командой чтения считывается N символов. Если на входе осталось меньше N символов, в буфер помещается специальный символ (eof). На буфер указывают два указателя: *продвижение* и *начало*. Между указателями размещается текущая лексема. Вначале они оба указывают на первый символ выделяемой лексемы. Один из них, *продвижение*, продвигается вперед, пока не будет выделена лексема, и

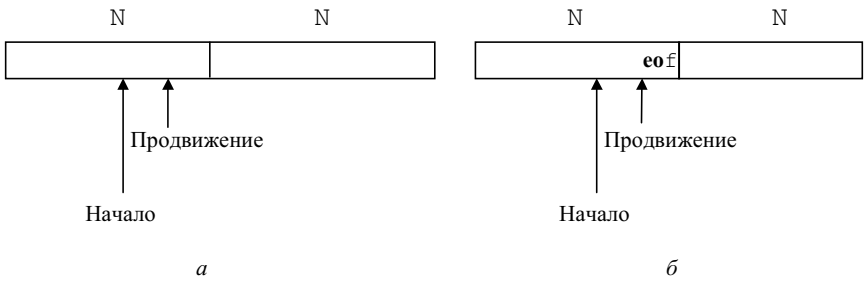


Рис. 3.18:

устанавливается на ее конец. После обработки лексемы оба указателя устанавливаются на символ, следующий за лексемой. Если указатель *продвижение* переходит середину буфера, правая половина заполняется новыми N символами. Если указатель *продвижение* переходит правую границу буфера, левая половина заполняется N символами и указатель *продвижение* устанавливается на начало буфера.

При каждом продвижении указателя необходимо проверять, не достигли ли мы границы одной из половин буфера. Для всех символов, кроме лежащих в конце половин буфера, требуются две проверки. Число проверок можно свести к одной, если в конце каждой половины поместить дополнительный "сторожевой" символ, в качестве которого логично взять eof (рис. 3.18, б).

В этом случае почти для всех символов делается единственная проверка на совпадение с eof и только в случае совпадения нужно дополнительно проверить, достигли ли мы середины или правого конца.

3.6 Конструктор лексических анализаторов LEX

Для автоматизации разработки лексических анализаторов было разработано довольно много средств. Как правило, входным языком для них служат либо праволинейные грамматики, либо язык регулярных выражений. Одной из наиболее распространенных систем является LEX, работающий с расширенными регулярными выражениями. LEX-программа состоит из трех частей:

```
Объявления
%%
Правила трансляции
%%
Вспомогательные подпрограммы
```

Секция объявлений включает объявления переменных, констант и определения регулярных выражений. При определении регулярных выражений могут использоваться следующие конструкции:

[abc]	– либо a, либо b, либо c;
[a-z]	– диапазон символов;
R*	– 0 или более повторений регулярного выражения R;
R+	– 1 или более повторений регулярного выражения R;
R1/R2	– R_1, если за ним следует R_2;
R1|R2	– либо R_1, либо R_2;
R?	– если есть R, выбрать его;
R$	– выбрать R, если оно последнее в строке;
^R	– выбрать R, если оно первое в строке;
[^R]	– дополнение к R;
R{n,m}	– повторение R от n до m раз;
{имя}	– именованное регулярное выражение;
(R)	– группировка.

Правила трансляции LEX программ имеют вид

```
p_1 { действие_1 }
p_2 { действие_2 }
.................
p_n { действие_n }
```

где каждое p_i – регулярное выражение, а каждое действие_i – фрагмент программы, описывающий, какое действие должен сделать лексический анализатор, когда образец p_i сопоставляется лексеме. В LEX действия записываются на Си.

Третья секция содержит вспомогательные процедуры, необходимые для действий. Эти процедуры могут транслироваться раздельно и загружаться с лексическим анализатором.

Лексический анализатор, сгенерированный LEX, взаимодействует с синтаксическим анализатором следующим образом. При вызове его синтаксическим анализатором лексический анализатор посимвольно читает остаток входа, пока не находит самый длинный префикс, который может быть сопоставлен одному из регулярных выражений p_i. Затем он выполняет действие_i. Как правило, действие_i возвращает управление синтаксическому анализатору. Если это не так, т.е. в соответствующем действии нет возврата, то лексический анализатор продолжает поиск лексем до тех, пока действие не вернет управление синтаксическому анализатору. Повторный поиск лексем вплоть до явной передачи управления позволяет лексическому анализатору правильно обрабатывать пробелы и комментарии. Синтаксическому анализатору лексический анализатор возвращает единственное значение – тип лексемы. Для передачи информации о типе лексемы используется глобальная переменная yylval. Текстовое представление выделенной лексемы хранится в переменной yytext, а ее длина в переменной yylen.

Пример 3.11. LEX-программа для ЛА, обрабатывающего идентификаторы, числа, ключевые слова if, then, else и знаки логических операций.

```
%{ /*определения констант LT,LE,EQ,NE,GT,
  GE,IF,THEN,ELSE,ID,NUMBER,RELOP, например,
  через DEFINE или скалярный тип*/ %}
 /*регулярные определения*/
 delim    [ \t\n]
 ws       {delim}+
 letter   [A-Za-z]
 digit    [0-9]
 id       {letter}({letter}|{digit})*
 number   {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
 %%
 {ws}     {/* действий и возврата нет */}
 if       {return(IF);}
 then     {return(THEN);}
 else     {return(ELSE);}
 {id}     {yylval=install_id(); return(ID);}
 {number} {yylval=install_num(); return(NUMBER);}
 "<"      {yylval=LT; return(RELOP);}
 "<="     {yylval=LE; return(RELOP);}
 "="      {yylval=EQ; return(RELOP);}
 "<>"     {yylval=NE; return(RELOP);}
 ">"      {yylval=GT; return(RELOP);}
 ">="     {yylval=GE; return(RELOP);}
 %%
 install_id()
 {/*подпрограмма, которая помещает лексему,
      на первый символ которой указывает yytext,
      длина которой равна yylen, в таблицу
      символов и возвращает указатель на нее*/
 }
 install_num()
 {/*аналогичная подпрограмма для размещения
          лексемы числа*/
 }
```

В разделе объявлений, заключенном в скобки %{ и %}, перечислены константы, используемые правилами трансляции. Все, что заключено в эти скобки, непосредственно копируется в программу лексического анализатора lex.yy.c и не рассматривается как часть регулярных определений или правил трансляции. То же касается и вспомогательных подпрограмм третьей секции. В данном примере это подпрограммы install_id и install_num.

В секцию определений входят также некоторые регулярные определения. Каждое такое определение состоит из имени и регулярного выражения, обозначаемого этим именем. Например, первое определенное имя – это delim. Оно обозначает класс символов { \t\n\}, т.е. любой из трех символов: пробел, табуляция или новая строка. Второе определение – разделитель, обозначаемый именем ws. Разделитель – это любая последовательность одного или более символов-

разделителей. Слово delim должно быть заключено в скобки, чтобы отличить его от образца, состоящего из пяти символов delim.

В определении letter используется класс символов. Сокращение [A-Za-z] означает любую из прописных букв от A до Z или строчных букв от a до z. В пятом определении для id для группировки используются скобки, являющиеся метасимволами LEX. Аналогично, вертикальная черта – метасимвол LEX, обозначающий объединение.

В последнем регулярном определении number символ "+" используется как метасимвол "одно или более вхождений", символ "?" как метасимвол "ноль или одно вхождение". Обратная черта используется для того, чтобы придать обычный смысл символу, использующемуся в LEX как метасимвол. В частности, десятичная точка в определении number обозначается как "\.", поскольку точка сама по себе представляет класс, состоящий из всех символов, за исключением символа новой строки. В классе символов [+\] обратная черта перед минусом стоит потому, что знак минус используется как символ диапазона, как в [A-Z].

Если символ имеет смысл метасимвола, то придать ему обычный смысл можно и по-другому, заключив его в кавычки. Так, в секции правил трансляции шесть операций отношения заключены в кавычки.

Рассмотрим правила трансляции, следующие за первым %%. Согласно первому правилу, если обнаружено ws, т.е. максимальная последовательность пробелов, табуляций и новых строк, никаких действий не производится. В частности, не осуществляется возврат в синтаксический анализатор.

Согласно второму правилу, если обнаружена последовательность букв "if", нужно вернуть значение IF, которое определено как целая константа, понимаемая синтаксическим анализатором как лексема "if". Аналогично обрабатываются ключевые слова "then" и "else" в двух следущих правилах.

В действии, связанном с правилом для id, два оператора. Переменной yylval присваивается значение, возвращаемое процедурой install_id. Переменная yylval определена в программе lex.yy.c, выходе LEX, и она доступна синтаксическому анализатору. yylval хранит возвращаемое лексическое значение, поскольку второй оператор в действии, return(ID), может только возвратить код класса лексем. Функция install_id заносит идентификаторы в таблицу символов.

Аналогично обрабатываются числа в следующем правиле. В последних шести правилах yylval используется для возврата кода операции отношения, возвращаемое же функцией значение – это код лексемы relop.

Если, например, в текущий момент лексический анализатор обрабатывает лексему "if", то этой лексеме соответствуют два образца: "if" и {id} и более длинной строки, соответствующей образцу, нет. Поскольку образец "if" предшествует образцу для идентификатора, конфликт разрешается в пользу ключевого слова. Такая стратегия разрешения конфликтов позволяет легко резервировать ключевые слова.

Если на входе встречается "<=", то первому символу соответствует образец "<", но это не самый длинный образец, который соответствует префиксу входа. Стратегия выбора самого длинного префикса легко разрешает такого рода конфликты.

Глава 4

Синтаксический анализ

4.1 КС-грамматики и МП-автоматы

Пусть $G = (N, T, P, S)$ – контекстно-свободная грамматика. Введем несколько важных понятий и определений.

Вывод, в котором в любой сентенциальной форме на каждом шаге делается подстановка самого левого нетерминала, называется *левосторонним*. Если $S \Rightarrow^* u$ в процессе левостороннего вывода, то u – *левая сентенциальная форма*. Аналогично определяется *правосторонний* вывод. Будем обозначать шаги левого (правого) вывода посредством $\Rightarrow_l$ ($\Rightarrow_r$).

Упорядоченным графом называется пара (V, E), где V есть множество вершин, а E – множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид $((v, v_1), (v, v_2), \ldots, (v, v_n))$. Этот элемент указывает, что из вершины v выходят n дуг, причем первой из них считается дуга, входящая в вершину v_1, второй – дуга, входящая в вершину v_2, и т.д.

Упорядоченным помеченным деревом называется упорядоченный граф (V, E), основой которого является дерево и для которого определена функция $f : V \to F$ (функция разметки) для некоторого множества F.

Упорядоченное помеченное дерево D называется *деревом вывода* (или *деревом разбора*) цепочки w в КС-грамматике $G = (N, T, P, S)$, если выполнены следующие условия:

(1) корень дерева D помечен S;

(2) каждый лист помечен либо $a \in T$, либо e;

(3) каждая внутренняя вершина помечена нетерминалом $A \in N$;

(4) если X – нетерминал, которым помечена внутренняя вершина и $X_1, \ldots, X_n$ – метки ее прямых потомков в указанном порядке, то $X \to X_1 \ldots X_k$ – правило из множества P;

(5) Цепочка, составленная из выписанных слева направо меток листьев, равна w.

Грамматика G называется *неоднозначной*, если существует цепочка w, для которой имеется два или более различных деревьев вывода в G.

Грамматика G называется *леворекурсивной*, если в ней имеется нетерминал A такой, что существует вывод $A \Rightarrow^+ A\alpha$ для некоторой цепочки α.

Автомат с магазинной памятью (МП-автомат) – это семерка $M = (Q, T, \Gamma, D, q_0, Z_0, F)$, где

(1) Q – конечное множество *состояний*, представляющих всевозможные состояния управляющего устройства;

(2) T – конечный *входной* алфавит;

(3) Γ – конечный алфавит *магазинных символов*;

(4) D – отображение множества $Q \times (T \cup \{e\}) \times \Gamma$ в множество всех конечных подмножеств $Q \times \Gamma^*$, называемое *функцией переходов*;

(5) $q_0 \in Q$ – *начальное состояние* управляющего устройства;

(6) $Z_0 \in \Gamma$ – символ, находящийся в магазине в начальный момент (*начальный символ магазина*);

(7) $F \subseteq Q$ – множество *заключительных* состояний.

Конфигурацией МП-автомата называется тройка (q, w, u), где

(1) $q \in Q$ – текущее состояние управляющего устройства;

(2) $w \in T^*$ – непрочитанная часть входной цепочки; первый символ цепочки w находится под входной головкой; если $w = e$, то считается, что вся входная лента прочитана;

(3) $u \in \Gamma^*$ – содержимое магазина; самый левый символ цепочки u считается верхним символом магазина; если $u = e$, то магазин считается пустым.

Такт работы МП-автомата M будем представлять в виде бинарного отношения $\vdash$, определенного на конфигурациях. Будем писать

$$(q, aw, Zu) \vdash (p, w, vu),$$

если множество $D(q, a, Z)$ содержит (p, v), где $q, p \in Q$, $a \in T \cup \{e\}$, $w \in T^*$, $Z \in \Gamma$ и $u, v \in \Gamma^*$.

Начальной конфигурацией МП-автомата M называется конфигурация вида (q_0, w, Z_0), где $w \in T^*$, т.е. управляющее устройство находится в начальном состоянии, входная лента содержит цепочку, которую

нужно проанализировать, а в магазине находится только начальный символ Z_0.

Заключительная конфигурация – это конфигурация вида (q, e, u), где $q \in F$, $u \in \Gamma^*$, т.е. управляющее устройство находится в одном из заключительных состояний, а входная цепочка целиком прочитана.

Введем транзитивное и рефлексивно-транзитивное замыкание отношения $\vdash$, а также его степень $k \geqslant 0$ (обозначаемые $\vdash^+$, $\vdash^*$ и $\vdash^k$ соответственно).

Говорят, что цепочка w *допускается* МП-автоматом M, если $(q_0, w, Z_0) \vdash^* (q, e, u)$ для некоторых $q \in F$ и $u \in \Gamma^*$.

Язык, *допускаемый (распознаваемый, определяемый)* автоматом M (обозначается $L(M)$) – это множество всех цепочек, допускаемых автоматом M.

Пример 4.1. Рассмотрим МП-автомат

$$M = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z, a, b\}, D, q_0, Z, \{q_2\}),$$

у которого функция переходов D содержит следующие элементы:

$D(q_0, a, Z) = \{(q_0, aZ)\}$,
$D(q_0, b, Z) = \{(q_0, bZ)\}$,
$D(q_0, a, a) = \{(q_0, aa), \{q_1, e)\}$,
$D(q_0, a, b) = \{(q_0, ab)\}$,
$D(q_0, b, a) = \{(q_0, ba)\}$,
$D(q_0, b, b) = \{(q_0, bb), (q_1, e)\}$,
$D(q_1, a, a) = \{(q_1, e)\}$,
$D(q_1, b, b) = \{(q_1, e)\}$,
$D(q_1, e, Z) = \{(q_2, e)\}$.

Нетрудно показать, что $L(M) = \{ww^R | w \in \{a, b\}^+\}$, где w^R обозначает обращение ("переворачивание") цепочки w.

Иногда допустимость определяют несколько иначе: цепочка w допускается МП-автоматом M, если $(q_0, w, Z_0) \vdash^* (q, e, e)$ для некоторого $q \in Q$. В таком случае говорят, что автомат допускает цепочку *опустошением магазина*. Эти определения эквивалентны, ибо справедлива

Теорема 4.1. *Язык допускается магазинным автоматом тогда и только тогда, когда он допускается (некоторым другим автоматом) опустошением магазина.*

Доказательство. Пусть $L = L(M)$ для некоторого МП-автомата $M = (Q, T, \Gamma, D, q_0, Z_0, F)$. Построим новый МП-автомат M', допускающий тот же язык опустошением магазина.

Пусть $M' = (Q \cup \{q_0', q_e\}, T, \Gamma \cup \{Z_0'\}, D', q_0', Z_0', \varnothing)$, где функция переходов D' определена следующим образом:

1. Если $(r, u) \in D(q, a, Z)$, то $(r, u) \in D'(q, a, Z)$ для всех $q \in Q$, $a \in T \cup \{e\}$ и $Z \in \Gamma$;

2. $D'(q'_0, e, Z'_0) = \{(q_0, Z_0 Z'_0)\}$;
3. Для всех $q \in F$ и $Z \in \Gamma \cup \{Z'_0\}$ множество $D'(q, e, Z)$ содержит (q_e, e);
4. $D'(q_e, e, Z) = \{(q_e, e)\}$ для всех $Z \in \Gamma \cup \{Z'_0\}$.

Автомат сначала переходит в конфигурацию $(q_0, w, Z_0 Z'_0)$ в соответствии с определением D' в п.2, затем в $(q, e, Y_1 \ldots Y_k Z'_0)$, $q \in F$ в соответствии с п.1, затем в $(q_e, e, Y_1 \ldots Y_k Z'_0)$, $q \in F$ в соответствии с п.3, затем в (q_e, e, e) в соответствии с п.4. Нетрудно показать по индукции, что $(q_0, w, Z_0) \vdash^+ (q, e, u)$ (где $q \in F$) выполняется для автомата M тогда и только тогда, когда $(q'_0, w, Z'_0) \vdash^+ (q_e, e, e)$ выполняется для автомата M'. Поэтому $L(M) = L'$, где L' – язык, допускаемый автоматом M' опустошением магазина.

Обратно, пусть $M = (Q, T, \Gamma, D, q_0, Z_0, \varnothing)$ – МП-автомат, допускающий опустошением магазина язык L. Построим автомат M', допускающий тот же язык по заключительному состоянию.

Пусть $M' = (Q \cup \{q'_0, q_f\}, T, \Gamma \cup \{X\}, D', q'_0, Z'_0, \{q_f\})$, где D' определяется следующим образом:

1. $D'(q'_0, e, Z'_0) = \{(q_0, Z_0 Z'_0)\}$ – переход в "режим M";
2. Для каждого $q \in Q$, $a \in T \cup \{e\}$, и $Z \in \Gamma$ определим $D'(q, a, Z) = D(q, a, Z)$ – работа в "режиме M";
3. Для всех $q \in Q$, $(q_f, e) \in D'(q, e, Z'_0)$ – переход в заключительное состояние.

Нетрудно показать по индукции, что $L = L(M')$. ■

Одним из важнейших результатов теории контекстно-свободных языков является доказательство эквивалентности МП-автоматов и КС-грамматик.

Теорема 4.2. *Язык является контекстно-свободным тогда и только тогда, когда он допускается магазинным автоматом.*

Доказательство. Пусть $G = (N, T, P, S)$ – КС-грамматика. Построим МП-автомат M, допускающий язык $L(G)$ опустошением магазина.

Пусть $M = (\{q\}, T, N \cup T, D, q, S, \varnothing)$, где D определяется следующим образом:

1. Если $A \to u \in P$, то $(q, u) \in D(q, e, A)$;
2. $D(q, a, a) = \{(q, e)\}$ для всех $a \in T$.

Фактически, этот МП-автомат в точности моделирует все возможные выводы в грамматике G. Нетрудно показать по индукции, что для любой цепочки $w \in T^*$ вывод $S \Rightarrow^+ w$ в грамматике G существует тогда и

только тогда, когда существует последовательность тактов $(q, w, S) \vdash^+ (q, e, e)$ автомата M.

Обратно, пусть $M = (Q, T, \Gamma, D, q_0, Z_0, \varnothing)$ – МП-автомат, допускающий опустошением магазина язык L. Построим грамматику G, порождающую язык L.

Пусть $G = (\{\ [qZr] \mid q, r \in Q, Z \in \Gamma\} \cup \{S\}, T, P, S)$, где P состоит из правил следующего вида:

1. Если $(r, X_1 \dots X_k) \in D(q, a, Z)$, $k \geqslant 1$, то

$$[qZs_k] \to a[rX_1s_1][s_1X_2s_2] \dots [s_{k-1}X_ks_k]$$

для любого набора $s_1, s_2, \dots, s_k$ состояний из Q;

2. Если $(r, e) \in D(q, a, Z)$, то $[qZr] \to a \in P$, $a \in T \cup \{e\}$;

3. $S \to [q_0Z_0q] \in P$ для всех $q \in Q$.

Нетерминалы и правила вывода грамматики определены так, что работе автомата M при обработке цепочки w соответствует левосторонний вывод w в грамматике G.

Индукцией по числу шагов вывода в G или числу тактов M нетрудно показать, что $(q, w, A) \vdash^+ (p, e, e)$ тогда и только тогда, когда $[qAp] \Rightarrow^+ w$.

Тогда, если $w \in L(G)$, то $S \Rightarrow [q_0Z_0q] \Rightarrow^+ w$ для некоторого $q \in Q$. Следовательно, $(q_0, w, Z_0) \vdash^+ (q, e, e)$ и поэтому $w \in L$. Аналогично, если $w \in L$, то $(q_0, w, Z_0) \vdash^+ (q, e, e)$. Значит, $S \Rightarrow [q_0Z_0q] \Rightarrow^+ w$, и поэтому $w \in L(G)$.

■

МП-автомат $M = (Q, T, \Gamma, D, q_0, Z_0, F)$ называется *детерминированным* (ДМП-автоматом), если выполнены следующие два условия:

(1) Множество $D(q, a, Z)$ содержит не более одного элемента для любых $q \in Q$, $a \in T \cup \{e\}$, $Z \in \Gamma$;

(2) Если $D(q, e, Z) \neq \varnothing$, то $D(q, a, Z) = \varnothing$ для всех $a \in T$.

Язык, допускаемый ДМП-автоматом, называется *детерминированным* КС-языком.

Так как функция переходов ДМП-автомата содержит не более одного элемента для любой тройки аргументов, мы будем пользоваться записью $D(q, a, Z) = (p, u)$ для обозначения $D(q, a, Z) = \{(p, u)\}$.

Пример 4.2. Рассмотрим ДМП-автомат

$$M = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{Z, a, b\}, D, q_0, Z, \{q_2\}),$$

у которого функция переходов определяется следующим образом:

$D(q_0, X, Y) = (q_0, XY)$, $X \in \{a, b\}$, $Y \in \{Z, a, b\}$,
$D(q_0, c, Y) = (q_1, Y)$, $Y \in \{a, b\}$,
$D(q_1, X, X) = (q_1, e)$, $X \in \{a, b\}$,
$D(q_1, e, Z) = (q_2, e)$.

Нетрудно показать, что этот детерминированный МП-автомат допускает язык $L = \{wcw^R | w \in \{a, b\}^+\}$.

К сожалению, ДМП-автоматы имеют меньшую распознавательную способность, чем МП-автоматы. Доказано, в частности, что существуют КС-языки, не являющиеся детерминированными КС-языками (таковым, например, является язык из примера 4.1).

Рассмотрим еще одну важную разновидность МП-автомата.

Расширенным автоматом с магазинной памятью назовем семерку $M = (Q, T, \Gamma, D, q_0, Z_0, F)$, где смысл всех символов тот же, что и для обычного МП-автомата, кроме D, представляющего собой отображение конечного подмножества множества $Q \times (T \cup \{e\}) \times \Gamma^*$ во множество конечных подмножеств множества $Q \times \Gamma^*$. Все остальные определения (конфигурации, такта, допустимости) для расширенного МП-автомата остаются такими же, как для обычного.

Теорема 4.3. *Пусть $M = (Q, T, \Gamma, D, q_0, Z_0, F)$ – расширенный МП-автомат. Тогда существует такой МП-автомат M', что $L(M') = L(M)$.*

Расширенный МП-автомат $M = (Q, T, \Gamma, D, q_0, Z_0, F)$ называется *детерминированным*, если выполнены следующие условия:

(1) Множество $D(q, a, u)$ содержит не более одного элемента для любых $q \in Q$, $a \in T \cup \{e\}$, $Z \in \Gamma^*$;

(2) Если $D(q, a, u) \neq \varnothing$, $D(q, a, v) \neq \varnothing$ и $u \neq v$, то не существует цепочки x такой, что $u = vx$ или $v = ux$;

(3) Если $D(q, a, u) \neq \varnothing$, $D(q, e, v) \neq \varnothing$, то не существует цепочки x такой, что $u = vx$ или $v = ux$.

Теорема 4.4. *Пусть $M = (Q, T, \Gamma, D, q_0, Z_0, F)$ – расширенный ДМП-автомат. Тогда существует такой ДМП-автомат M', что $L(M') = L(M)$.*

ДМП-автомат и расширенный ДМП-автомат лежат в основе рассматриваемых далее в этой главе, соответственно, LL и LR-анализаторов.

4.2 Преобразования КС-грамматик

Рассмотрим ряд преобразований, позволяющих “улучшить” вид контекстно-свободной грамматики, без изменения порождаемого ею языка.

Назовем символ $X \in (N \cup T)$ *недостижимым* в КС-грамматике $G = (N, T, P, S)$, если X не появляется ни в одной выводимой цепочке этой грамматики. Иными словами, символ X является недостижимым, если в G не существует вывода $S \Rightarrow^* \alpha X \beta$ ни для каких $\alpha, \beta \in (N \cup T)^*$.

Алгоритм 4.1. Устранение недостижимых символов.

Вход. КС-грамматика $G = (N, T, P, S)$.

Выход. КС-грамматика $G' = (N', T', P', S)$ без недостижимых символов, такая, что $L(G') = L(G)$.

Метод. Выполнить шаги 1–4:

(1) Положить $V_0 = \{S\}$ и $i = 1$.

(2) Положить $V_i = \{X \mid$ в P есть $A \to \alpha X \beta$ и $A \in V_{i-1}\} \cup V_{i-1}$.

(3) Если $V_i \neq V_{i-1}$, положить $i = i + 1$ и перейти к шагу 2. В противном случае перейти к шагу 4.

(4) Положить $N' = V_i \cap N$, $T' = V_i \cap T$. Включить в P' все правила из P, содержащие только символы из V_i.

Назовем символ $X \in (N \cup T)$ *бесполезным* в КС-грамматике $G = (N, T, P, S)$, если в ней не существует вывода вида $S \Rightarrow^* xXy \Rightarrow^* xwy$, где w, x, y принадлежат T^*. Очевидно, что каждый недостижимый символ является бесполезным.

Алгоритм 4.2. Устранение бесполезных символов.

Вход. КС-грамматика $G = (N, T, P, S)$.

Выход. КС-грамматика $G' = (N', T', P', S)$ без бесполезных символов, такая, что $L(G') = L(G)$.

Метод. Выполнить шаги 1–5:

(1) Положить $N_0 = \varnothing$ и $i = 1$.

(2) Положить $N_i = \{A | A \to \alpha \in P$ и $\alpha \in (N_{i-1} \cup T)^*\} \cup N_{i-1}$.

(3) Если $N_i \neq N_{i-1}$, положить $i = i+1$ и перейти к шагу 2. В противном случае положить $N_e = N_i$ и перейти к шагу 4.

(4) Положить $G_1 = ((N \cap N_e) \cup \{S\}, T, P_1, S)$, где P_1 состоит из правил множества P, содержащих только символы из $N_e \cup T$.

(5) Применив к G_1 алгоритм 4.1, получить $G' = (N', T', P', S)$.

КС-грамматика без бесполезных символов называется *приведенной*. Легко видеть, что для любой КС-грамматики существует эквивалентная приведенная. В дальнейшем будем предполагать, что все рассматриваемые грамматики – приведенные.

4.3 Предсказывающий разбор сверху-вниз

4.3.1 Алгоритм разбора сверху-вниз

Пусть дана КС-грамматика $G = (N, T, P, S)$. Рассмотрим *предсказывающий разбор* (или *разбор сверху-вниз*) для грамматики G.

Основная задача предсказывающего разбора – определение правила вывода, которое нужно применить к нетерминалу. Процесс предсказывающего разбора с точки зрения построения дерева разбора проиллюстрирован на рис. 4.1.

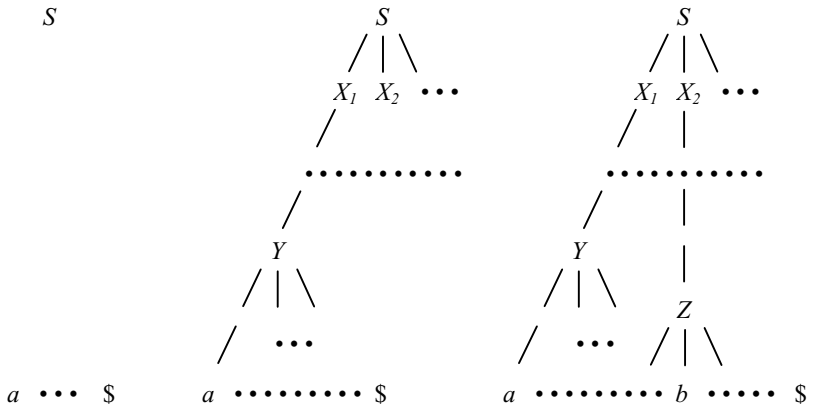


Рис. 4.1:

Фрагменты недостроенного дерева соответствуют сентенциальным формам. Вначале дерево состоит только из одной вершины, соответствующей аксиоме S. В этот момент по первому символу входной цепочки предсказывающий анализатор должен определить правило $S \to X_1 X_2 \ldots$, которое должно быть применено к S. Затем необходимо определить правило, которое должно быть применено к X_1, и т.д., до тех пор, пока в процессе такого построения сентенциальной формы, соответствующей левому выводу, не будет применено правило $Y \to a \ldots$. Этот процесс затем применяется для следующего самого левого нетерминального символа сентенциальной формы.

На рис. 4.2 приведена структура предсказывающего анализатора, который определяет очередное правило с помощью таблицы. Такую таблицу можно построить непосредственно по грамматике.

Таблично-управляемый предсказывающий анализатор имеет входную ленту, управляющее устройство (программу), таблицу анализа, магазин (стек) и выходную ленту.

Входная лента содержит анализируемую строку, заканчивающуюся символом \$ – маркером конца строки. Выходная лента содержит после-

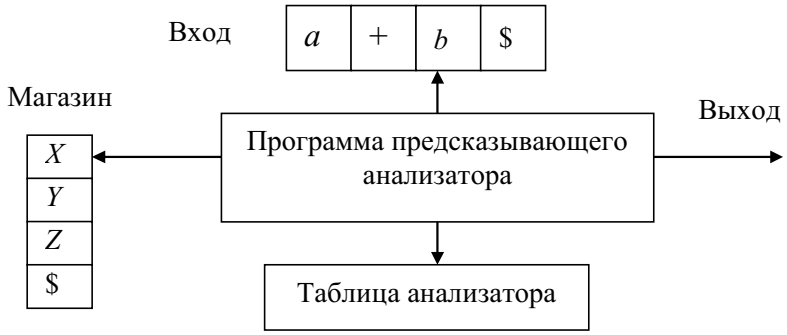


Рис. 4.2:

довательность примененных правил вывода.

Таблица анализа – это двумерный массив $M[A, a]$, где A – нетерминал, и a – терминал или символ \$. Значением $M[A, a]$ может быть некоторое правило грамматики или элемент “ошибка”.

Магазин может содержать последовательность символов грамматики с \$ на дне. В начальный момент магазин содержит только начальный символ грамматики на верхушке и \$ на дне.

Анализатор работает следующим образом. Вначале анализатор находится в конфигурации, в которой магазин содержит S\$, на входной ленте w\$ (w – анализируемая цепочка), выходная лента пуста. На каждом такте анализатор рассматривает X – символ на верхушке магазина и a – текущий входной символ. Эти два символа определяют действия анализатора. Имеются следующие возможности.

1. Если $X = a =$ \$, анализатор останавливается, сообщает об успешном окончании разбора и выдает содержимое выходной ленты.

2. Если $X = a \neq$ \$, анализатор удаляет X из магазина и продвигает указатель входа на следующий входной символ.

3. Если X – терминал, и $X \neq a$, то анализатор останавливается и сообщает о том, что входная цепочка не принадлежит языку.

4. Если X – нетерминал, анализатор заглядывает в таблицу $M[X, a]$. Возможны два случая:

а) Значением $M[X, a]$ является правило для X. В этом случае анализатор заменяет X на верхушке магазина на правую часть данного правила, а само правило помещает на выходную ленту. Указатель входа не передвигается.

б) Значением $M[X, a]$ является “ошибка”. В этом случае анализатор останавливается и сообщает о том, что входная цепочка не принадлежит языку.

Работа анализатора может быть задана следующей программой:

```
do
  {X=верхний символ магазина;
   if (X - терминал || X=="$")
      if (X==InSym)
         {удалить X из магазина;
          InSym=очередной символ;
         }
      else error();
   else /*X - нетерминал*/
      if (M[X,InSym]=="X->Y1Y2...Yk")
         {удалить X из магазина;
          поместить Yk,Yk-1,...Y1 в магазин
            (Y1 на верхушку);
          вывести правило X->Y1Y2...Yk;
         }
      else error(); /*вход таблицы M пуст*/
  }
while (X!=$) /*магазин пуст*/
```

Пример 4.3. Рассмотрим грамматику арифметических выражений $G = (\{E, E', T, T', F\}, \{id, +, *, (,)\}, P, E)$ с правилами:

$$\begin{array}{ll} E \to TE' & T' \to *FT' \\ E' \to +TE' & T' \to e \\ E' \to e & F \to (E) \\ T \to FT' & F \to id \end{array}$$

Таблица предсказывающего анализатора для этой грамматики приведена на рис. 4.3. Пустые клетки таблицы соответствуют элементу “ошибка”.

Нетерминал	Входной символ					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to e$	$E' \to e$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to e$	$T' \to *FT'$		$T' \to e$	$T' \to e$
F	$F \to id$			$F \to (E)$		

Рис. 4.3:

При разборе входной цепочки $id + id * id$\$ анализатор совершает последовательность шагов, изображенную на рис. 4.4. Заметим, что анализатор осуществляет в точности левый вывод. Если за уже просмотренными входными символами поместить символы грамматики в магазине, то можно получить в точности левые сентенциальные формы вывода. Дерево разбора для этой цепочки приведено на рис. 4.5.

Магазин	Вход	Выход
$E\$$	$id + id * id\$$	
$TE'\$$	$id + id * id\$$	$E \to TE'$
$FT'E'\$$	$id + id * id\$$	$T \to FT'$
$id\,T'E'\$$	$id + id * id\$$	$F \to id$
$T'E'\$$	$+id * id\$$	
$E'\$$	$+id * id\$$	$T' \to e$
$+TE'\$$	$+id * id\$$	$E' \to +TE$
$TE'\$$	$id * id\$$	
$FT'E'\$$	$id * id\$$	$T \to FT'$
$id\,T'E'\$$	$id * id\$$	$F \to id$
$T'E'\$$	$*id\$$	
$*F'T'E'\$$	$*id\$$	$T' \to *FT'$
$FT'E'\$$	$id\$$	
$id\,T'E'\$$	$id\$$	$F \to id$
$T'E'\$$	$\$$	
$E'\$$	$\$$	$T' \to e$
$\$$	$\$$	$E' \to e$

Рис. 4.4:

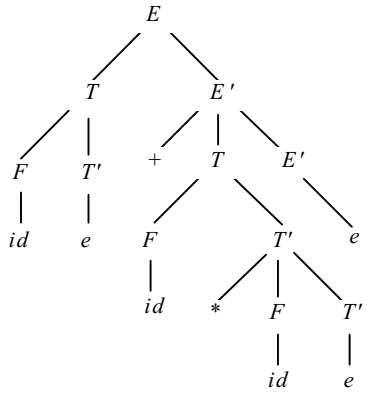


Рис. 4.5:

4.3.2 Функции $FIRST$ и $FOLLOW$

При построении таблицы предсказывающего анализатора нам потребуются две функции – $FIRST$ и $FOLLOW$.

Пусть $G = (N, T, P, S)$ – КС-грамматика. Для α – произвольной цепочки, состоящей из символов грамматики, определим $FIRST(\alpha)$ как множество терминалов, с которых начинаются строки, выводимые из α. Если $\alpha \Rightarrow^* e$, то e также принадлежит $FIRST(\alpha)$.

Определим $FOLLOW(A)$ для нетерминала A как множество терминалов a, которые могут появиться непосредственно справа от A в некоторой сентенциальной форме грамматики, т.е. множество терминалов a таких, что существует вывод вида $S \Rightarrow^* \alpha A a \beta$ для некоторых $\alpha, \beta \in (N \cup T)^*$. Заметим, что между A и a в процессе вывода могут находиться нетерминальные символы, из которых выводится e. Если A может быть самым правым символом некоторой сентенциальной формы, то \$ также принадлежит $FOLLOW(A)$.

Рассмотрим алгоритмы вычисления функции $FIRST$.

Алгоритм 4.3. Вычисление $FIRST$ для символов грамматики.
Вход. КС-грамматика $G = (N, T, P, S)$.
Выход. Множество $FIRST(X)$ для каждого символа $X \in (N \cup T)$.
Метод. Выполнить шаги 1–3:

(1) Если X – терминал, то положить $FIRST(X) = \{X\}$;
если X – нетерминал, положить $FIRST(X) = \varnothing$.

(2) Если в P имеется правило $X \to e$, то добавить e к $FIRST(X)$.

(3) Пока ни к какому множеству $FIRST(X)$ нельзя уже будет добавить новые элементы, выполнять:

если X – нетерминал и имеется правило вывода $X \to Y_1 Y_2 \ldots Y_k$, то включить a в $FIRST(X)$, если $a \in FIRST(Y_i)$ для некоторого i, $1 \leqslant i \leqslant k$, и e принадлежит всем $FIRST(Y_1), \ldots, FIRST(Y_{i-1})$, то есть $Y_1 \ldots Y_{i-1} \Rightarrow^* e$. Если e принадлежит $FIRST(Y_j)$ для всех $j = 1, 2, \ldots, k$, то добавить e к $FIRST(X)$.

Алгоритм 4.4. Вычисление $FIRST$ для цепочки.
Вход. КС-грамматика $G = (N, T, P, S)$.
Выход. Множество $FIRST(X_1 X_2 \ldots X_n)$, $X_i \in (N \cup T)$.
Метод. Выполнить шаги 1–3:

(1) При помощи предыдущего алгоритма вычислить $FIRST(X)$ для каждого $X \in (N \cup T)$.

(2) Положить $FIRST(X_1 X_2 \ldots X_n) = \varnothing$.

(3) Добавить к $FIRST(X_1 X_2 \ldots X_n)$ все не e-элементы из $FIRST(X_1)$. Добавить к нему также все не e-элементы из $FIRST(X_2)$, если $e \in FIRST(X_1)$, не e-элементы из $FIRST(X_3)$, если e принадлежит как $FIRST(X_1)$, так и $FIRST(X_2)$, и т.д. Наконец, добавить цепочку e к $FIRST(X_1 X_2 \ldots X_n)$, если $e \in FIRST(X_i)$ для всех i.

Рассмотрим алгоритм вычисления функции $FOLLOW$.

Алгоритм 4.5. Вычисление $FOLLOW$ для нетерминалов грамматики.

Вход. КС-грамматика $G = (N, T, P, S)$.

Выход. Множество $FOLLOW(X)$ для каждого символа $X \in N$.

Метод. Выполнить шаги 1–4:

(1) Положить $FOLLOW(X) = \varnothing$ для каждого символа $X \in N$.

(2) Добавить \$ к $FOLLOW(S)$.

(3) Если в P есть правило вывода $A \to \alpha B \beta$, где $\alpha, \beta \in (N \cup T)^*$ то все элементы из $FIRST(\beta)$, за исключением e, добавить к $FOLLOW(B)$.

(4) Пока ничего нельзя будет добавить ни к какому множеству $FOLLOW(X)$, выполнять:

если в P есть правило $A \to \alpha B$ или $A \to \alpha B \beta$, $\alpha, \beta \in (N \cup T)^*$, где $FIRST(\beta)$ содержит e (т.е. $\beta \Rightarrow^* e$), то все элементы из $FOLLOW(A)$ добавить к $FOLLOW(B)$.

Пример 4.4. Рассмотрим грамматику из примера 4.3. Для нее

$$
\begin{aligned}
&FIRST(E) = FIRST(T) = FIRST(F) = \{(, id\} \\
&FIRST(E') = \{+, e\} \\
&FIRST(T') = \{*, e\} \\
&FOLLOW(E) = FOLLOW(E') = \{\ ), \$\} \\
&FOLLOW(T) = FOLLOW(T') = \{+,), \$\} \\
&FOLLOW(F) = \{+, *,), \$\}
\end{aligned}
$$

Например, id и левая скобка добавляются к $FIRST(F)$ на шаге 3 при $i = 1$, поскольку $FIRST(id) = \{id\}$ и $FIRST(\text{"("}) = \{\text{"("}\}$ в соответствии с шагом 1. На шаге 3 при $i = 1$, в соответствии с правилом вывода $T \to FT'$ к $FIRST(T)$ добавляются также id и левая скобка. На шаге 2 в $FIRST(E')$ включается e.

При вычислении множеств $FOLLOW$ на шаге 2 в $FOLLOW(E)$ включается \$. На шаге 3, на основании правила $F \to (E)$, к $FOLLOW(E)$ добавляется также правая скобка. На шаге 4, примененном к правилу $E \to TE'$, в $FOLLOW(E')$ включаются \$ и правая скобка. Поскольку $E' \Rightarrow^* e$, они также попадают и во множество $FOLLOW(T)$. В соответствии с правилом вывода $E \to TE'$, на шаге 3 в $FOLLOW(T)$ включаются и все элементы из $FIRST(E')$, отличные от e.

4.3.3 Конструирование таблицы предсказывающего анализатора

Для конструирования таблицы предсказывающего анализатора по грамматике G может быть использован алгоритм, основанный на следующей идее. Предположим, что $A \to \alpha$ – правило вывода грамматики и $a \in FIRST(\alpha)$. Тогда анализатор делает развертку A по α, если входным символом является a. Трудность возникает, когда $\alpha = e$ или $\alpha \Rightarrow^* e$. В этом случае нужно развернуть A в α, если текущий входной символ принадлежит $FOLLOW(A)$ или если достигнут \$ и $\$ \in FOLLOW(A)$.

Алгоритм 4.6. Построение таблицы предсказывающего анализатора.

Вход. КС-грамматика $G = (N, T, P, S)$.

Выход. Таблица $M[A, a]$ предсказывающего анализатора, $A \in N$, $a \in T \cup \{\$\}$.

Метод. Для каждого правила вывода $A \to \alpha$ грамматики выполнить шаги 1 и 2. После этого выполнить шаг 3.

(1) Для каждого терминала a из $FIRST(\alpha)$ добавить $A \to \alpha$ к $M[A, a]$.

(2) Если $e \in FIRST(\alpha)$, добавить $A \to \alpha$ к $M[A, b]$ для каждого терминала b из $FOLLOW(A)$. Кроме того, если $e \in FIRST(\alpha)$ и $\$ \in FOLLOW(A)$, добавить $A \to \alpha$ к $M[A, \$]$.

(3) Положить все неопределенные входы равными "ошибка".

Пример 4.5. Применим алгоритм 4.6 к грамматике из примера 4.3. Поскольку $FIRST(TE') = FIRST(T) = \{(, id\ \}$, в соответствии с правилом вывода $E \to TE'$ входы $M[E, (\ ]$ и $M[E, id\ ]$ становятся равными $E \to TE'$.

В соответствии с правилом вывода $E' \to +TE'$ значение $M[E', +]$ равно $E' \to +TE'$. В соответствии с правилом вывода $E' \to e$ значения $M[E',)]$ и $M[E', \$]$ равны $E' \to e$, поскольку $FOLLOW(E') = \{\ ), \$\}$.

Таблица анализа, построенная алгоритмом 4.6 для этой грамматики, приведена на рис. 4.3.

4.3.4 LL(1)-грамматики

Алгоритм 4.6 для построения таблицы предсказывающего анализатора может быть применен к любой КС-грамматике. Однако для некоторых грамматик построенная таблица может иметь неоднозначно определенные входы. Нетрудно доказать, например, что если грамматика леворекурсивна или неоднозначна, таблица будет иметь по крайней мере один неоднозначно определенный вход.

Грамматики, для которых таблица предсказывающего анализатора не имеет неоднозначно определенных входов, называются LL(1)-грамматиками. Предсказывающий анализатор, построенный для LL(1)-грамматики, называется LL(1)-анализатором. Первая буква L в названии связано с тем, что входная цепочка читается слева направо, вторая L означает, что строится левый вывод входной цепочки, 1 – что на каждом шаге для принятия решения используется один символ непрочитанной части входной цепочки.

Доказано, что алгоритм 4.6 для каждой LL(1)-грамматики G строит таблицу предсказывающего анализатора, распознающего все цепочки из $L(G)$ и только эти цепочки. Нетрудно доказать также, что если G – LL(1)-грамматика, то $L(G)$ – детерминированный КС-язык.

Справедлив также следующий критерий LL(1)-грамматики. Грамматика $G = (N, T, P, S)$ является LL(1)-грамматикой тогда и только тогда,

когда для каждой пары правил $A \to \alpha$, $A \to \beta$ из P (т.е. правил с одинаковой левой частью) выполняются следующие 2 условия:

(1) $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$;

(2) Если $e \in FIRST(\alpha)$, то $FIRST(\beta) \cap FOLLOW(A) = \varnothing$.

Язык, для которого существует порождающая его LL(1)-грамматика, называют LL(1)-языком. Доказано, что проблема определения того, порождает ли грамматика LL-язык, является алгоритмически неразрешимой.

Пример 4.6. Неоднозначная грамматика не является LL(1). Примером может служить следующая грамматика $G = (\{S, E\}, \{if, then, else, a, b\}, P, S)$ с правилами:

$S \to if\ E\ then\ S \mid if\ E\ then\ S\ else\ S \mid a$

$E \to b$

Эта грамматика является неоднозначной, что иллюстрируется на рис. 4.6.

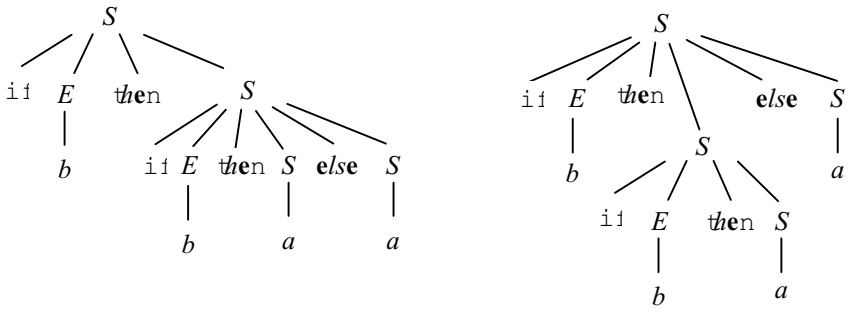


Рис. 4.6:

4.3.5 Удаление левой рекурсии

Основная трудность при использовании предсказывающего анализа – это нахождение такой грамматики для входного языка, по которой можно построить таблицу анализа с однозначно определенными входами. Иногда с помощью некоторых простых преобразований грамматику, не являющуюся LL(1), можно привести к эквивалентной LL(1)-грамматике. Среди этих преобразований наиболее эффективными являются левая факторизация и удаление левой рекурсии. Здесь необходимо сделать два замечания. Во-первых, не всякая грамматика после этих преобразований становится LL(1), и, во-вторых, после таких преобразований получающаяся грамматика может стать менее понимаемой.

Непосредственную левую рекурсию, т.е. рекурсию вида $A \to A\alpha$, можно удалить следующим способом. Сначала группируем A-правила:

$$A \to A\alpha_1 \mid A\alpha_2 \mid \ \ldots \ \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ \ldots \ \mid \beta_n,$$

где никакая из строк β_i не начинается с A. Затем заменяем этот набор правил на

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ \ldots \ \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ \ldots \ \mid \alpha_m A' \mid e$$

где A' – новый нетерминал. Из нетерминала A можно вывести те же цепочки, что и раньше, но теперь нет левой рекурсии. С помощью этой процедуры удаляются все непосредственные левые рекурсии, но не удаляется левая рекурсия, включающая два или более шага. Приведенный ниже алгоритм 4.7 позволяет удалить все левые рекурсии из грамматики.

Алгоритм 4.7. Удаление левой рекурсии.
Вход. КС-грамматика G без e-правил (правил вида $A \to e$).
Выход. КС-грамматика G' без левой рекурсии, эквивалентная G.
Метод. Выполнить шаги 1 и 2.

(1) Упорядочить нетерминалы грамматики G в произвольном порядке.

(2) Выполнить следующую процедуру:

```
for (i=1;i<=n;i++){
  for (j=1;j<=i-1;j++){
    пусть Aj → β1 | β2 | ... | βk - все текущие правила для Aj;
    заменить все правила вида Ai → Ajα
    на правила Ai → β1α | β2α | ... | βkα;
  }
  удалить правила вида Ai → Ai;
  удалить непосредственную левую рекурсию в правилах для Ai;
}
```

После $(i-1)$-й итерации внешнего цикла на шаге 2 для любого правила вида $A_k \to A_s\alpha$, где $k < i$, выполняется $s > k$. В результате на следующей итерации (по i) внутренний цикл (по j) последовательно увеличивает нижнюю границу по m в любом правиле $A_i \to A_m\alpha$, пока не будет $m \geqslant i$. Затем, после удаления непосредственной левой рекурсии для A_i-правил, m становится больше i.

Алгоритм 4.7 применим, если грамматика не имеет e-правил (правил вида $A \to e$). Имеющиеся в грамматике e-правила могут быть удалены предварительно. Получающаяся грамматика без левой рекурсии может иметь e-правила.

4.3.6 Левая факторизация

Основная идея левой факторизации в том, что в том случае, когда неясно, какую из двух альтернатив надо использовать для развертки нетерминала A, нужно изменить A-правила так, чтобы отложить решение до тех пор, пока не будет достаточно информации для принятия правильного решения.

Если $A \to \alpha\beta_1 \mid \alpha\beta_2$ – два A-правила и входная цепочка начинается с непустой строки, выводимой из α, мы не знаем, разворачивать ли по первому правилу или по второму. Можно отложить решение, развернув $A \to \alpha A'$. Тогда после анализа того, что выводимо из α, можно развернуть по $A' \to \beta_1$ или по $A' \to \beta_2$. Левофакторизованные правила принимают вид

$$A \to \alpha A'$$

$$A' \to \beta_1 \mid \beta_2$$

Алгоритм 4.8. Левая факторизация грамматики.

Вход. КС-грамматика G.

Выход. Левофакторизованная КС-грамматика G', эквивалентная G.

Метод. Для каждого нетерминала A ищем самый длинный префикс α, общий для двух или более его альтернатив. Если $\alpha \neq e$, т.е. существует нетривиальный общий префикс, заменяем все A-правила

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ \ldots \ \mid \alpha\beta_n \mid z,$$

где z обозначает все альтернативы, не начинающиеся с α, на

$$A \to \alpha A' \mid z$$

$$A' \to \beta_1 \mid \beta_2 \mid \ \ldots \ \mid \beta_n$$

где A' – новый нетерминал. Повторно применяем это преобразование, пока никакие две альтернативы не будут иметь общего префикса.

Пример 4.7. Рассмотрим вновь грамматику условных операторов из примера 4.6:

$S \to if\ E\ then\ S \mid if\ E\ then\ S\ else\ S \mid a$
$E \to b$

После левой факторизации грамматика принимает вид

$S \to if\ E\ then\ SS' \mid a$
$S' \to else\ S \mid e$
$E \to b$

К сожалению, грамматика остается неоднозначной, а значит, и не LL(1).

4.3.7 Рекурсивный спуск

Выше был рассмотрен таблично-управляемый вариант предсказывающего анализа, когда магазин явно использовался в процессе работы анализатора. Можно предложить другой вариант предсказывающего анализа, в котором каждому нетерминалу сопоставляется процедура (вообще говоря, рекурсивная), и магазин образуется неявно при вызовах таких процедур. Процедуры рекурсивного спуска могут быть записаны, как показано ниже.

В процедуре A для случая, когда имеется правило $A \to u_i$, такое, что $u_i \Rightarrow^* e$ (напомним, что не может быть больше одного правила, из которого выводится e), приведены два варианта 1.1 и 1.2. В варианте 1.1 делается проверка, принадлежит ли следующий входной символ $FOLLOW(A)$. Если нет – выдается ошибка. Во втором варианте этого не делается, так что анализ ошибки откладывается на процедуру, вызвавшую A.

```
void A(){ // A → u_1 | u_2 | ... | u_k
  if (InSym ∈ FIRST(u_i)) // только одному!
    if (parse(u_i))
      result("A → u_i");
    else error();
  else
  //Вариант 1:
  if (имеется правило A → u_i такое, что u_i ⇒* e)
    //Вариант 1.1
    if (InSym ∈ FOLLOW(A))
      result("A → u_i");
    else error();
    //Конец варианта 1.1
    //Вариант 1.2:
    result("A → u_i");
    //Конец варианта 1.2
  //Конец варианта 1
  //Вариант 2:
  if (нет правила A → u_i такого, что u ⇒* e)
    error();
  //Конец варианта 2
}

boolean parse(u){ // из u не выводится e!
  v = u;
  while (v ≠ e){ // v = Xz
    if (X - терминал a)
      if (InSym ≠ a)
        return(false);
      else InSym = getInsym();
```

```
        else // X - нетерминал B
          B;
          v = z;
      }
      return(true);
    }
```

4.3.8 Восстановление после синтаксических ошибок

В приведенных программах рекурсивного спуска использовалась процедура реакции на синтаксические ошибки error(). В простейшем случае эта процедура выдает диагностику и завершает работу анализатора. Но можно попытаться некоторым разумным образом продолжить работу. Для разбора сверху вниз можно предложить следующий простой алгоритм.

Если в момент обнаружения ошибки на верхушке магазина оказался нетерминальный символ A и для него нет правила, соответствующего входному символу, то сканируем вход до тех пор, пока не встретим символ либо из $FIRST(A)$, либо из $FOLLOW(A)$. В первом случае разворачиваем A по соответствующему правилу, во втором – удаляем A из магазина.

Если на верхушке магазина терминальный символ, то можно удалить все терминальные символы с верхушки магазина вплоть до первого (сверху) нетерминального символа и продолжать так, как это было описано выше.

4.4 Разбор снизу-вверх типа сдвиг-свертка

4.4.1 Основа

В процессе разбора снизу-вверх типа сдвиг-свертка строится дерево разбора входной цепочки, начиная с листьев (снизу) к корню (вверх). Этот процесс можно рассматривать как "свертку" цепочки w к начальному символу грамматики. На каждом шаге свертки подцепочка, которую можно сопоставить правой части некоторого правила вывода, заменяется символом левой части этого правила вывода, и если на каждом шаге выбирается правильная подцепочка, то в обратном порядке прослеживается правосторонний вывод (рис. 4.7). Здесь ко входной цепочке, так же как и при анализе LL(1)-грамматик, приписан концевой маркер $.

Подцепочка сентенциальной формы, которая может быть сопоставлена правой части некоторого правила вывода, свертка по которому к левой части правила соответствует одному шагу в обращении правостороннего вывода, называется *основой* цепочки. Самая левая подцепочка, которая сопоставляется правой части некоторого правила вывода $A \rightarrow$

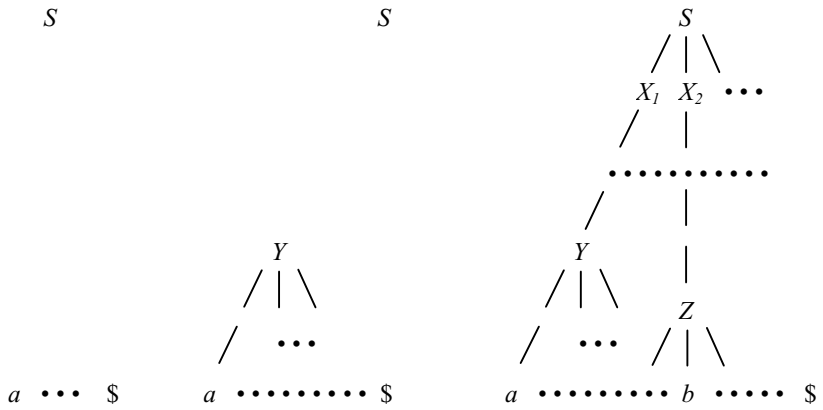


Рис. 4.7:

γ, не обязательно является основой, поскольку свертка по правилу $A \to \gamma$ может дать цепочку, которая не может быть сведена к аксиоме.

Формально, основа правой сентенциальной формы z – это правило вывода $A \to \gamma$ и позиция в z, в которой может быть найдена цепочка γ такие, что в результате замены γ на A получается предыдущая сентенциальная форма в правостороннем выводе z. Таким образом, если $S \Rightarrow_r^* \alpha A\beta \Rightarrow_r \alpha\gamma\beta$, то $A \to \gamma$ в позиции, следующей за α, это основа цепочки $\alpha\gamma\beta$. Подцепочка β справа от основы содержит только терминальные символы.

Вообще говоря, грамматика может быть неоднозначной, поэтому не единственным может быть правосторонний вывод $\alpha\gamma\beta$ и не единственной может быть основа. Если грамматика однозначна, то каждая правая сентенциальная форма грамматики имеет в точности одну основу. Замена основы в сентенциальной форме на нетерминал левой части называется *отсечением* основы. Обращение правостороннего вывода может быть получено с помощью повторного применения отсечения основы, начиная с исходной цепочки w. Если w – слово в рассматриваемой грамматике, то $w = \alpha_n$, где α_n – n-я правая сентенциальная форма еще неизвестного правого вывода $S = \alpha_0 \Rightarrow_r \alpha_1 \Rightarrow_r \ldots \Rightarrow_r \alpha_{n-1} \Rightarrow_r \alpha_n = w$.

Чтобы восстановить этот вывод в обратном порядке, выделяем основу γ_n в α_n и заменяем γ_n на левую часть некоторого правила вывода $A_n \to \gamma_n$, получая $(n-1)$-ю правую сентенциальную форму α_{n-1}. Затем повторяем этот процесс, т.е. выделяем основу γ_{n-1} в α_{n-1} и сворачиваем эту основу, получая правую сентенциальную форму α_{n-2}. Если, повторяя этот процесс, мы получаем правую сентенциальную форму, состоящую только из начального символа S, то останавливаемся и сообщаем об успешном завершении разбора. Обращение последовательности правил, использованных в свертках, есть правый вывод входной строки.

Таким образом, главная задача анализатора типа сдвиг-свертка – это выделение и отсечение основы.

4.4.2 LR(1)-анализаторы

В названии LR(1) символ L указывает на то, что входная цепочка читается слева-направо, R – на то, что строится правый вывод, наконец, 1 указывает на то, что анализатор видит один символ непрочитанной части входной цепочки.

LR(1)-анализ привлекателен по нескольким причинам:

- LR(1)-анализ – наиболее мощный метод анализа без возвратов типа сдвиг-свертка;
- LR(1)-анализ может быть реализован довольно эффективно;
- LR(1)-анализаторы могут быть построены для практически всех конструкций языков программирования;
- класс грамматик, которые могут быть проанализированы LR(1)-методом, строго включает класс грамматик, которые могут быть проанализированы предсказывающими анализаторами (сверху-вниз типа LL(1)).

Схематически структура LR(1)-анализатора изображена на рис. 4.8. Анализатор состоит из входной ленты, выходной ленты, магазина, управляющей программы и таблицы анализа (LR(1)-таблицы), которая имеет две части – *функцию действий* ($Action$) и *функцию переходов* ($Goto$). Управляющая программа одна и та же для всех LR(1)-анализаторов, разные анализаторы отличаются только таблицами анализа.

Программа анализатора читает символы на входной ленте по одному за шаг. В процессе анализа используется магазин, в котором хранятся строки вида $S_0X_1S_1X_2S_2 \ldots X_mS_m$ (S_m – верхушка магазина). Каждый X_i – символ грамматики (терминальный или нетерминальный), а S_i – символ *состояния*.

Заметим, что символы грамматики (либо символы состояний) не обязательно должны размещаться в магазине. Однако, их использование облегчает понимание поведения LR-анализатора.

Элемент функции действий $Action[S_m, a_i]$ для символа состояния S_m и входа $a_i \in T \cup \{\$\}$, может иметь одно из четырех значений:

1) shift S (сдвиг), где S – символ состояния,
2) reduce $A \to \gamma$ (свертка по правилу грамматики $A \to \gamma$),
3) accept (допуск),
4) error (ошибка).

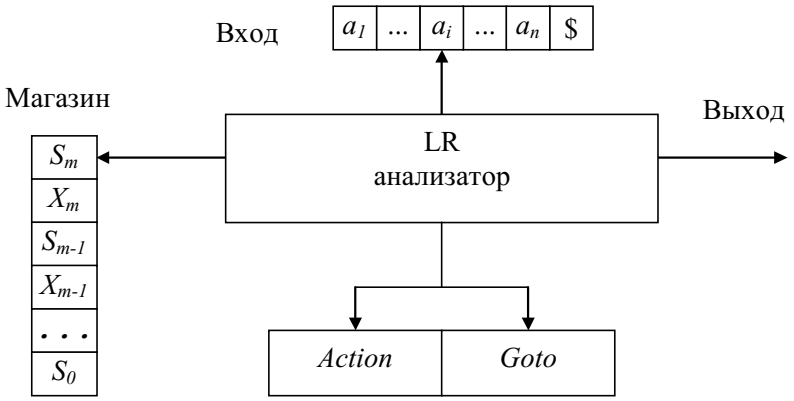


Рис. 4.8:

Элемент функции переходов $Goto[S_m, A]$ для символа состояния S_m и входа $A \in N$, может иметь одно из двух значений:

1) S, где S – символ состояния,

2) error (ошибка).

Конфигурацией LR(1)-анализатора называется пара, первая компонента которой – содержимое магазина, а вторая – непросмотренный вход:

$$(S_0X_1S_1X_2S_2 \ldots X_mS_m, a_ia_{i+1} \ldots a_n\$)$$

Эта конфигурация соответствует правой сентенциальной форме

$$X_1X_2 \ldots X_ma_ia_{i+1} \ldots a_n$$

Префиксы правых сентенциальных форм, которые могут появиться в магазине анализатора, называются *активными* префиксами. Основа сентенциальной формы всегда располагается на верхушке магазина. Таким образом, активный префикс – это такой префикс правой сентенциальной формы, который не переходит правую границу основы этой формы.

В начале работы анализатора в магазине находится только символ начального состояния S_0, на входной ленте – анализируемая цепочка с маркером конца.

Очередной шаг анализатора определяется текущим входным символом a_i и символом состояния на верхушке магазина S_m следующим образом.

Пусть LR(1)-анализатор находится в конфигурации

$$(S_0X_1S_1X_2S_2 \ldots X_mS_m, a_ia_{i+1} \ldots a_n\$)$$

Анализатор может проделать один из следующих шагов:

1. Если $Action[S_m, a_i] = \text{shift } S$, то анализатор выполняет сдвиг, переходя в конфигурацию

$$(S_0X_1S_1X_2S_2 \ldots X_mS_ma_iS, a_{i+1} \ldots a_n\$)$$

Таким образом, в магазин помещаются входной символ a_i и символ состояния S, определяемый $Action[S_m, a_i]$. Текущим входным символом становится a_{i+1}.

2. Если $Action[S_m, a_i] = \text{reduce } A \to \gamma$, то анализатор выполняет свертку, переходя в конфигурацию

$$(S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS, a_ia_{i+1} \ldots a_n\$)$$

где $S = Goto[S_{m-r}, A]$ и r – длина γ, правой части правила вывода.

Анализатор сначала удаляет из магазина $2r$ символов (r символов состояния и r символов грамматики), так что на верхушке оказывается состояние S_{m-r}. Затем анализатор помещает в магазин A – левую часть правила вывода, и S – символ состояния, определяемый $Goto[S_{m-r}, A]$. На шаге свертки текущий входной символ не меняется. Для LR(1)-анализаторов $X_{m-r+1} \ldots X_m$ – последовательность символов грамматики, удаляемых из магазина, всегда соответствует γ – правой части правила вывода, по которому делается свертка.

После осуществления шага свертки генерируется выход LR(1)-анализатора, т.е. исполняются семантические действия, связанные с правилом, по которому делается свертка, например, печатаются номера правил, по которым делается свертка.

Заметим, что функция $Goto$ таблицы анализа, построенная по грамматике G, фактически представляет собой функцию переходов детерминированного конечного автомата, распознающего активные префиксы G.

3. Если $Action[S_m, a_i] = \text{accept}$, то разбор успешно завершен.

4. Если $Action[S_m, a_i] = \text{error}$, то анализатор обнаружил ошибку, и выполняются действия по диагностике и восстановлению.

Пример 4.8. Рассмотрим грамматику арифметических выражений $G = (\{E, T, F\}, \{id, +, *\}, P, E)$ с правилами:

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to id$

На рис. 4.9 изображены функции *Action* и *Goto*, образующие LR(1)-таблицу для этой грамматики. Для Элемент Si функции *Action* означает сдвиг и помещение в магазин состояния с номером i, Rj – свертку по правилу номер j, acc – допуск, пустая клетка – ошибку. Для функции *Goto* символ i означает помещение в магазин состояния с номером i, пустая клетка – ошибку.

На входе $id+id*id$ последовательность состояний магазина и входной ленты показаны на рис. 4.10. Например, в первой строке LR-анализатор находится в нулевом состоянии и "видит" первый входной символ id. Действие S6 в нулевой строке и столбце id в поле *Action* (рис. 4.9) означает сдвиг и помещение символа состояния 6 на верхушку магазина. Это и изображено во второй строке: первый символ id и символ состояния 6 помещаются в магазин, а id удаляется со входной ленты.

Состояния	*Action*				*Goto*		
	id	+	*	$	E	T	F
0	S6				1	2	3
1		S4		acc			
2		R2	S7	R2			
3		R4	R4	R4			
4	S6					5	3
5		R1	S7	R1			
6		R5	R5	R5			
7	S6						8
8		R3	R3	R3			

Рис. 4.9:

Активный префикс	Магазин	Вход	Действие
	0	$id+id*id$\$	сдвиг
id	0 id 6	$+id*id$\$	$F \to id$
F	0 F 3	$+id*id$\$	$T \to F$
T	0 T 2	$+id*id$\$	$E \to T$
E	0 E 1	$+id*id$\$	сдвиг
$E+$	0 E 1 + 4	$id*id$\$	сдвиг
$E+id$	0 E 1 + 4 id 6	$*id$\$	$F \to id$
$E+F$	0 E 1 + 4 F 3	$*id$\$	$T \to F$
$E+T$	0 E 1 + 4 T 5	id\$	сдвиг
$E+T*$	0 E 1 + 4 T 5 * 7	id\$	сдвиг
$E+T*id$	0 E 1 + 4 T 5 * 7 id 6	\$	$F \to id$
$E+T*F$	0 E 1 + 4 T 5 * 7 F 8	\$	$T \to T*F$
$E+T$	0 E 1 + 4 T 5	\$	$E \to E+T$
E	0 E 1		допуск

Рис. 4.10:

Текущим входным символом становится $+$, и действием в состоянии 6 на вход $+$ является свертка по $F \to id$. Из магазина удаляются два символа (один символ состояния и один символ грамматики). Затем анализируется нулевое состояние. Поскольку $Goto$ в нулевом состоянии по символу F – это 3, F и 3 помещаются в магазин. Теперь имеем конфигурацию, соответствующую третьей строке. Остальные шаги определяются аналогично.

4.4.3 Конструирование LR(1)-таблицы

Рассмотрим теперь алгоритм конструирования таблицы, управляющей LR(1)-анализатором.

Пусть $G = (N, T, P, S)$ – КС-грамматика. *Пополненной* грамматикой для данной грамматики G называется КС-грамматика

$$G' = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S'),$$

т.е. эквивалентная грамматика, в которой введен новый начальный символ S' и новое правило вывода $S' \to S$.

Это дополнительное правило вводится для того, чтобы определить, когда анализатор должен остановить разбор и зафиксировать допуск входа. Таким образом, допуск имеет место тогда и только тогда, когда анализатор готов осуществить свертку по правилу $S' \to S$.

LR(1)-*ситуацией* называется пара $[A \to \alpha.\beta, a]$, где $A \to \alpha\beta$ – правило грамматики, a - терминал или правый концевой маркер \$. Вторая компонента ситуации называется *аванцепочкой*.

Будем говорить, что LR(1)-ситуация $[A \to \alpha.\beta, a]$ *допустима* для активного префикса δ, если существует вывод $S \Rightarrow_r^* \gamma Aw \Rightarrow_r \gamma\alpha\beta w$, где $\delta = \gamma\alpha$ и либо a – первый символ w, либо $w = e$ и $a =$ \$.

Будем говорить, что ситуация *допустима*, если она допустима для какого-либо активного префикса.

Пример 4.9. Рассмотрим грамматику $G = (\{S, B\}, \{a, b\}, P, S)$ с правилами
$S \to BB$
$B \to aB \mid b$

Существует правосторонний вывод $S \Rightarrow_r^* aaBab \Rightarrow_r aaaBab$. Легко видеть, что ситуация $[B \to a.B, a]$ допустима для активного префикса $\delta = aaa$, если в определении выше положить $\gamma = aa$, $A = B$, $w = ab$, $\alpha = a$, $\beta = B$. Существует также правосторонний вывод $S \Rightarrow_r^* BaB \Rightarrow_r BaaB$. Поэтому для активного префикса Baa допустима ситуация $[B \to a.B,$ \$$]$.

Центральная идея метода заключается в том, что по грамматике строится детерминированный конечный автомат, распознающий активные префиксы. Для этого ситуации группируются во множества, которые и образуют состояния автомата. Ситуации можно рассматривать как состояния недетерминированного конечного автомата, распознающего активные префиксы, а их группировка на самом деле есть процесс построения детерминированного конечного автомата из недетерминированного.

Анализатор, работающий слева-направо по типу сдвиг-свертка, должен уметь распознавать основы на верхушке магазина. Состояние автомата после прочтения содержимого магазина и текущий входной символ определяют очередное действие автомата. Функцией переходов этого конечного автомата является функция переходов LR-анализатора. Чтобы не просматривать магазин на каждом шаге анализа, на верхушке магазина всегда хранится то состояние, в котором должен оказаться этот конечный автомат после того, как он прочитал символы грамматики в магазине от дна к верхушке.

Рассмотрим ситуацию вида $[A \to \alpha.B\beta, a]$ из множества ситуаций, допустимых для некоторого активного префикса z. Тогда существует правосторонний вывод $S \Rightarrow_r^* yAax \Rightarrow_r y\alpha B\beta ax$, где $z = y\alpha$. Предположим, что из βax выводится терминальная строка bw. Тогда для некоторого правила вывода вида $B \to q$ имеется вывод $S \Rightarrow_r^* zBbw \Rightarrow_r zqbw$. Таким образом $[B \to .q, b]$ также допустима для z и ситуация $[A \to \alpha B.\beta, a]$ допустима для активного префикса zB. Здесь либо b может быть первым терминалом, выводимым из β, либо из β выводится e в выводе $\beta ax \Rightarrow_r^* bw$ и тогда b равно a. Т.е. b принадлежит $FIRST(\beta ax)$. Построение всех таких ситуаций для данного множества ситуаций, т.е. его замыкание, делает приведенная ниже функция closure.

Система множеств допустимых LR(1)-ситуаций для всевозможных активных префиксов пополненной грамматики называется *канонической системой* множеств допустимых LR(1)-ситуаций. Алгоритм построения канонической системы множеств приведен ниже.

Алгоритм 4.9. Конструирование канонической системы множеств допустимых LR(1)-ситуаций.

Вход. КС-грамматика $G = (N, T, P, S)$.

Выход. Каноническая система C множеств допустимых LR(1)-ситуаций для грамматики G.

Метод. Заключается в выполнении для пополненной грамматики G' процедуры items, которая использует функции closure и goto.

```
function closure(I){ /* I - множество ситуаций */
  do{
    for (каждой ситуации [A → α.Bβ, a] из I,
         каждого правила вывода B → γ из G',
         каждого терминала b из FIRST(βa),
         такого, что [B → .γ, b] нет в I)
      добавить [B → .γ, b] к I;
  }
  while (к I можно добавить новую ситуацию);
  return I;
}

function goto(I,X){ /* I - множество ситуаций;
```

```
                                    X - символ грамматики */
    Пусть J = {[A → αX.β, a] | [A → α.Xβ, a] ∈ I};
    return closure(J);
}

procedure items(G'){ /* G' - пополненная грамматика */
    I_0 = closure({[S' → .S, $]});
    C = {I_0};
    do{
      for (каждого множества ситуаций I из системы C,
           каждого символа грамматики X такого,
           что goto(I, X) не пусто и не принадлежит C)
        добавить goto(I, X) к системе C;
    }
    while (к C можно добавить новое множество ситуаций);
}
```

Если I – множество ситуаций, допустимых для некоторого активного префикса δ, то $\mathrm{goto}(I, X)$ – множество ситуаций, допустимых для активного префикса δX.

Работа алгоритма построения системы C множеств допустимых LR(1)-ситуаций начинается с того, что в C помещается *начальное* множество ситуаций $I_0 = \mathrm{closure}(\{[S' \to .S, \$]\})$. Затем с помощью функции goto вычисляются новые множества ситуаций и включаются в C. По-существу, $\mathrm{goto}(I, X)$ – переход конечного автомата из состояния I по символу X.

Рассмотрим теперь, как по системе множеств LR(1)-ситуаций строится LR(1)-таблица, т.е. функции действий и переходов LR(1)-анализатора.

Алгоритм 4.10. Построение LR(1)-таблицы.

Вход. Каноническая система $C = \{I_0, I_1, \ldots, I_n\}$ множеств допустимых LR(1)-ситуаций для грамматики G.

Выход. Функции $Action$ и $Goto$, составляющие LR(1)-таблицу для грамматики G.

Метод. Для каждого состояния i функции $Action[i, a]$ и $Goto[i, X]$ строятся по множеству ситуаций I_i:

(1) Значения функции действия ($Action$) для состояния i определяются следующим образом:

   а) если $[A \to \alpha.a\beta, b] \in I_i$ (a – терминал) и $\mathrm{goto}(I_i, a) = I_j$, то полагаем $Action[i, a] = \mathrm{shift}\ j$;

   б) если $[A \to \alpha., a] \in I_i$, причем $A \neq S'$, то полагаем $Action[i, a] = \mathrm{reduce}\ A \to \alpha$;

   в) если $[S' \to S., \$] \in I_i$, то полагаем $Action[i, \$] = \mathrm{accept}$.

(3) Значения функции переходов для состояния i определяются следующим образом: если $\mathtt{goto}(I_i, A) = I_j$, то $Goto[i, A] = j$ (здесь A – нетерминал).

(4) Все входы в $Action$ и $Goto$, не определенные шагами 2 и 3, полагаем равными error.

(5) Начальное состояние анализатора строится из множества, содержащего ситуацию $[S' \to .S, \$]$.

Таблица на основе функций $Action$ и $Goto$, полученных в результате работы алгоритма 4.10, называется *канонической* LR(1)-таблицей. LR(1)-анализатор, работающий с этой таблицей, называется *каноническим* LR(1)-анализатором.

Пример 4.10. Рассмотрим следующую грамматику, являющуюся пополненной для грамматики из примера 4.8:

(0) $E' \to E$
(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to id$

Множества ситуаций и переходы по goto для этой грамматики приведены на рис. 4.11. LR(1)-таблица для этой грамматики приведена на рис. 4.9.

4.4.4 LR(1)-грамматики

Если для КС-грамматики G функция $Action$, полученная в результате работы алгоритма 4.10, не содержит неоднозначно определенных входов, то грамматика называется LR(1)-грамматикой.

Язык L называется LR(1)-языком, если он может быть порожден некоторой LR(1)-грамматикой.

Иногда используется другое определение LR(1)-грамматики. Грамматика называется LR(1), если из условий

1. $S' \Rightarrow_r^* uAw \Rightarrow_r uvw$,
2. $S' \Rightarrow_r^* zBx \Rightarrow_r uvy$,
3. $FIRST(w) = FIRST(y)$

следует, что $uAy = zBx$ (т.е. $u = z$, $A = B$ и $x = y$).

Согласно этому определению, если uvw и uvy – правовыводимые цепочки пополненной грамматики, у которых $FIRST(w) = FIRST(y)$ и $A \to v$ – последнее правило, использованное в правом выводе цепочки uvw, то правило $A \to v$ должно применяться и в правом разборе при свертке uvy к uAy. Так как A дает v независимо от w, то LR(1)-условие означает, что в $FIRST(w)$ содержится информация, достаточная для определения того, что uv за один шаг выводится из uA. Поэтому никогда не

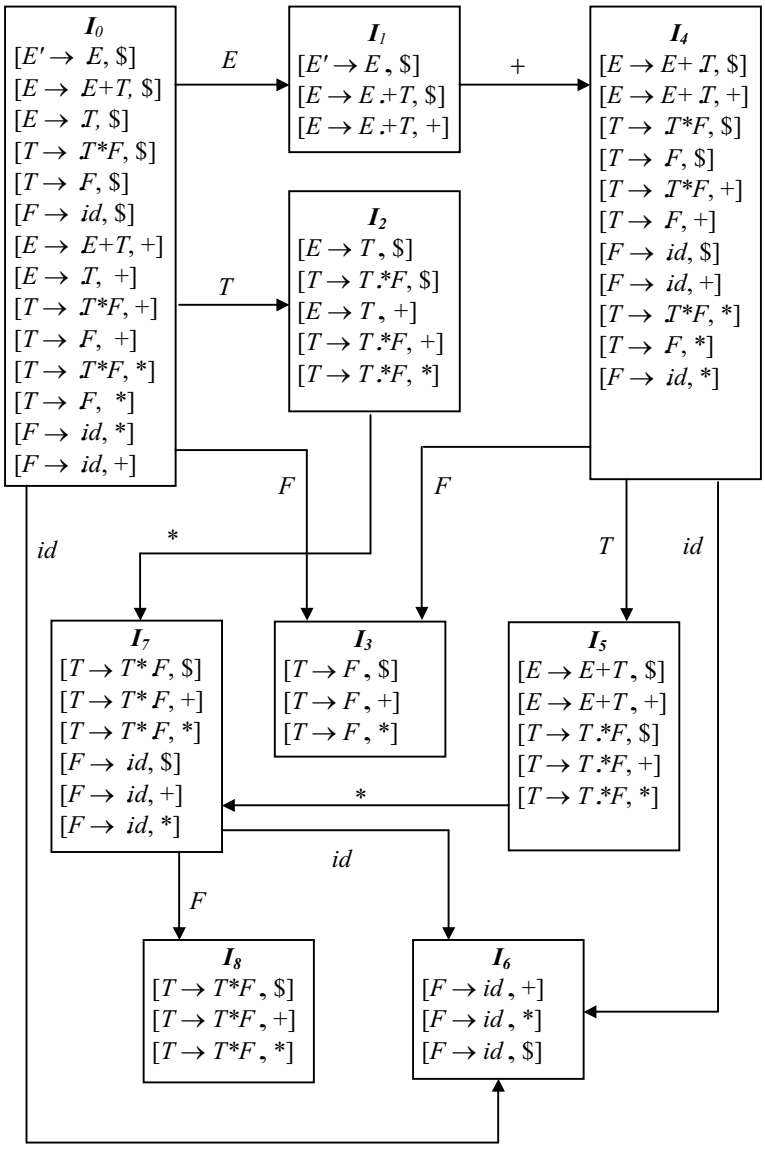


Рис. 4.11:

может возникнуть сомнений относительно того, как свернуть очередную правовыводимую цепочку пополненной грамматики.

Можно доказать, что эти два определения эквивалентны.

Если грамматика не является LR(1), то анализатор типа сдвиг-свертка при анализе некоторой цепочки может достигнуть конфигурации, в ко-

торой он, зная содержимое магазина и следующий входной символ, не может решить, делать ли сдвиг или свертку (конфликт сдвиг/свертка), или не может решить, какую из нескольких сверток применить (конфликт свертка/свертка).

В частности, неоднозначная грамматика не может быть LR(1). Для доказательства рассмотрим два различных правых вывода

(1) $S \Rightarrow_r u_1 \Rightarrow_r ... \Rightarrow_r u_n \Rightarrow_r w$, и

(2) $S \Rightarrow_r v_1 \Rightarrow_r ... \Rightarrow_r v_m \Rightarrow_r w$.

Нетрудно заметить, что LR(1)-условие (согласно второму определению LR(1)-грамматики) нарушается для наименьшего из чисел i, для которых $u_{n-i} \neq v_{m-i}$.

Пример 4.11. Рассмотрим вновь грамматику условных операторов:

$S \to if\ E\ then\ S \mid if\ E\ then\ S\ else\ S \mid a$
$E \to b$

Если анализатор типа сдвиг-свертка находится в конфигурации, такой что необработанная часть входной цепочки имеет вид $else$ ... \$, а в магазине находится ... $if\ E\ then\ S$, то нельзя определить, является ли $if\ E\ then\ S$ основой, вне зависимости от того, что лежит в магазине ниже. Это конфликт сдвиг/свертка. В зависимости от того, что следует на входе за $else$, правильной может быть свертка по $S \to if\ E\ then\ S$ или сдвиг $else$, а затем разбор другого S и завершение основы $if\ E\ then\ S\ else\ S$. Таким образом нельзя сказать, нужно ли в этом случае делать сдвиг или свертку, так что грамматика не является LR(1).

Эта грамматика может быть преобразована к LR(1)-виду следующим образом:

$S \to M \mid U$
$M \to if\ E\ then\ M\ else\ M \mid a$
$U \to if\ E\ then\ S \mid if\ E\ then\ M\ else\ U$
$E \to b$

Основная разница между LL(1)- и LR(1)-грамматиками заключается в следующем. Чтобы грамматика была LR(1), необходимо распознавать вхождение правой части правила вывода, просмотрев все, что выведено из этой правой части и текущий символ входной цепочки. Это требование существенно менее строгое, чем требование для LL(1)-грамматики, когда необходимо определить применимое правило, видя только первый символ, выводимый из его правой части. Таким образом, класс LL(1)-грамматик является собственным подклассом класса LR(1)-грамматик.

Справедливы также следующие утверждения [2].

Теорема 4.5. *Каждый LR(1)-язык является детерминированным КС-языком.*

Теорема 4.6. *Если L – детерминированный КС-язык, то существует LR(1)-грамматика, порождающая L.*

4.4.5 Восстановление после синтаксических ошибок

Одним из простейших методов восстановления после ошибки при LR(1)-анализе является следующий. При синтаксической ошибке просматриваем магазин от верхушки, пока не найдем состояние s с переходом на выделенный нетерминал A. Затем сканируются входные символы, пока не будет найден такой, который допустим после A. В этом случае на верхушку магазина помещается состояние $Goto[s, A]$ и разбор продолжается. Для нетерминала A может иметься несколько таких вариантов. Обычно A – это нетерминал, представляющий одну из основных конструкций языка, например оператор.

При более детальной проработке реакции на ошибки можно в каждой пустой клетке анализатора поставить обращение к своей подпрограмме. Такая подпрограмма может вставлять или удалять входные символы или символы магазина, менять порядок входных символов.

4.4.6 Варианты LR-анализаторов

Часто построенные таблицы для LR(1)-анализатора оказываются довольно большими. Поэтому при практической реализации используются различные методы их сжатия. С другой стороны, часто оказывается, что при построении для языка синтаксического анализатора типа "сдвиг-свертка" достаточно более простых методов. Некоторые из этих методов базируются на основе LR(1)-анализаторов.

Одним из способов такого упрощения является LR(0)-анализ – частный случая LR-анализа, когда ни при построении таблиц, ни при анализе не учитывается аванцепочка.

Еще одним вариантом LR-анализа являются так называемые SLR(1)-анализаторы (Simple LR(1)). Они строятся следующим образом. Пусть $C = \{I_0, I_1, \dots, I_n\}$ – набор множеств допустимых LR(0)-ситуаций. Состояния анализатора соответствуют I_i. Функции действий и переходов анализатора определяются следующим образом.

1. Если $[A \to u.av] \in I_i$ и $\mathtt{goto}(I_i, a) = I_j$, то определим $Action[i, a] =$ shift j.

2. Если $[A \to u.] \in I_i$, то определим $Action[i, a] =$ reduce $A \to u$ для всех $a \in FOLLOW(A)$, $A \neq S'$.

3. Если $[S' \to S.] \in I_i$, то определим $Action[i, \$] =$ accept.

4. Если $\mathtt{goto}(I_i, A) = I_j$, где $A \in N$, то определим $Goto[i, A] = j$.

5. Остальные входы для функций $Action$ и $Goto$ определим как error.

6. Начальное состояние соответствует множеству ситуаций, содержащему ситуацию $[S' \to .S]$.

Распространенным вариантом LR(1)-анализа является также LALR(1)-анализ. Он основан на объединении (слиянии) некоторых таблиц. Назовем *ядром* множества LR(1)-ситуаций множество их первых компонент (т.е. во множестве ситуаций не учитываются аванцепочки). Объединим все множества ситуаций с одинаковыми ядрами, а в качестве аванцепочек возьмем объединение аванцепочек. Функции $Action$ и $Goto$ строятся очевидным образом. Если функция $Action$ таким образом построенного анализатора не имеет конфликтов, то он называется LALR(1)-анализатором (LookAhead LR(1)).

Глава 5

Элементы теории перевода

До сих пор мы рассматривали процесс синтаксического анализа только как процесс анализа допустимости входной цепочки. Однако, в компиляторе синтаксический анализ служит основой еще одного важного шага – построения дерева синтаксического анализа. В примерах 4.3 и 4.8 предыдущей главы в процессе синтаксического анализа в качестве выхода выдавалась последовательность примененных правил, на основе которой и может быть построено дерево. Построение дерева синтаксического анализа является простейшим частным случаем *перевода* – процесса преобразования некоторой входной цепочки в некоторую выходную.

Определение. Пусть T – входной алфавит, а Π – выходной алфавит. *Переводом* (или *трансляцией*) с языка $L_1 \subseteq T^*$ на язык $L_2 \subseteq \Pi^*$ называется отображение $\tau : L_1 \to L_2$. Если $y = \tau(x)$, то цепочка y называется *выходом* для цепочки x.

Мы рассмотрим несколько формализмов для определения переводов: преобразователи с магазинной памятью, схемы синтаксически управляемого перевода и атрибутные грамматики.

5.1 Преобразователи с магазинной памятью

Рассмотрим важный класс абстрактных устройств, называемых преобразователями с магазинной памятью. Эти преобразователи получаются из автоматов с магазинной памятью, если к ним добавить выход и позволить на каждом шаге выдавать выходную цепочку.

Преобразователем с магазинной памятью (МП-преобразователем) называется восьмерка $P = (Q, T, \Gamma, \Pi, D, q_0, Z_0, F)$, где все символы имеют тот же смысл, что и в определении МП-автомата, за исключением того, что Π – конечный выходной алфавит, а D – отображение множе-

ства $Q \times (T \cup \{e\}) \times \Gamma$ в множество конечных подмножеств множества $Q \times \Gamma^* \times \Pi^*$.

Определим *конфигурацию* преобразователя P как четверку (q, x, u, y), где $q \in Q$ – состояние, $x \in T^*$ – цепочка на входной ленте, $u \in \Gamma^*$ – содержимое магазина, $y \in \Pi^*$ – цепочка на выходной ленте, выданная вплоть до настоящего момента.

Если множество $D(q, a, Z)$ содержит элемент (r, u, z), то будем писать $(q, ax, Zw, y) \vdash (r, x, uw, yz)$ для любых $x \in T^*$, $w \in \Gamma^*$ и $y \in \Pi^*$. Рефлексивно-транзитивное замыкание отношения $\vdash$ будем обозначать $\vdash^*$.

Цепочку y назовем *выходом* для x, если $(q_0, x, Z_0, e) \vdash^* (q, e, u, y)$ для некоторых $q \in F$ и $u \in \Gamma^*$. *Переводом* (или *трансляцией*), определяемым МП-преобразователем P (обозначается $\tau(P)$), назовем множество

$$\{(x, y) \mid (q_0, x, Z_0, e) \vdash^* (q, e, u, y) \text{ для некоторых } q \in F \text{ и } u \in \Gamma^*\}$$

Будем говорить, что МП-преобразователь P является *детерминированным* (ДМП-преобразователем), если выполняются следующие условия:

1) для всех $q \in Q$, $a \in T \cup \{e\}$ и $Z \in \Gamma$ множество $D(q, a, Z)$ содержит не более одного элемента,

2) если $D(q, e, Z) \neq \varnothing$, то $D(q, a, Z) = \varnothing$ для всех $a \in T$.

Пример 5.1. Рассмотрим перевод τ, отображающий каждую цепочку $x \in \{a, b\}^*\$$, в которой число вхождений символа a равно числу вхождений символа b, в цепочку $y = (ab)^n$, где n – число вхождений a или b в цепочку x. Например, $\tau(abbaab\$) = ababab$.

Этот перевод может быть реализован ДМП-преобразователем $P = (\{q_0, q_f\}, \{a, b, \$\}, \{Z, a, b\}, \{a, b\}, D, q_0, Z, \{q_f\})$ с функцией переходов:

$D(q_0, X, Z) = \{(q_0, XZ, e)\}$, $X \in \{a, b\}$,
$D(q_0, \$, Z) = \{(q_f, Z, e)\}$,
$D(q_0, X, X) = \{(q_0, XX, e)\}$, $X \in \{a, b\}$,
$D(q_0, X, Y) = \{(q_0, e, ab)\}$, $X \in \{a, b\}$, $Y \in \{a, b\}$, $X \neq Y$.

5.2 Синтаксически управляемый перевод

Другим формализмом, используемым для определения переводов, является схема синтаксически управляемого перевода. Фактически, такая схема представляет собой КС-грамматику, в которой к каждому правилу добавлен элемент перевода. Всякий раз, когда правило участвует в выводе входной цепочки, с помощью элемента перевода вычисляется часть выходной цепочки, соответствующая части входной цепочки, порожденной этим правилом.

5.2.1 Схемы синтаксически управляемого перевода

Определение. *Схемой синтаксически управляемого перевода* (или трансляции, сокращенно: СУ-схемой) называется пятерка $Tr = (N, T, \Pi, R, S)$, где

(1) N – конечное множество нетерминальных символов;

(2) T – конечный входной алфавит;

(3) Π – конечный выходной алфавит;

(4) R – конечное множество *правил перевода* вида

$$A \to u, v$$

где $u \in (N \cup T)^*$, $v \in (N \cup \Pi)^*$ и вхождения нетерминалов в цепочку v образуют перестановку вхождений нетерминалов в цепочку u, так что каждому вхождению нетерминала B в цепочку u соответствует некоторое вхождение этого же нетерминала в цепочку v; если нетерминал B встречается более одного раза, для указания соответствия используются верхние целочисленные индексы;

(5) S – начальный символ, выделенный нетерминал из N.

Определим *выводимую пару* в схеме Tr следующим образом:

(1) (S, S) – выводимая пара, в которой символы S соответствуют друг другу;

(2) если $(xAy, x'Ay')$ – выводимая пара, в цепочках которой вхождения A соответствуют друг другу, и $A \to u, v$ – правило из R, то $(xuy, x'vy')$ – выводимая пара. Для обозначения такого вывода одной пары из другой будем пользоваться обозначением $\Rightarrow$: $(xAy, x'Ay') \Rightarrow (xuy, x'vy')$. Рефлексивно-транзитивное замыкание отношение $\Rightarrow$ обозначим $\Rightarrow^*$.

Переводом $\tau(Tr)$, *определяемым* СУ-*схемой* Tr, назовем множество пар

$$\{(x, y) \mid (S, S) \Rightarrow^* (x, y),\ x \in T^*,\ y \in \Pi^*\}$$

Если через P обозначить множество входных правил вывода всех правил перевода, то $G = (N, T, P, S)$ будет *входной грамматикой* для Tr.

СУ-схема $Tr = (N, T, \Pi, R, S)$ называется *простой*, если для каждого правила $A \to u, v$ из R соответствующие друг другу вхождения нетерминалов встречаются в u и v в одном и том же порядке.

Перевод, определяемый простой СУ-схемой, называется *простым синтаксически управляемым переводом* (простым СУ-переводом).

Пример 5.2. Перевод арифметических выражений в ПОЛИЗ (польскую инверсную запись) можно осуществить простой СУ-схемой с правилами

$$\begin{array}{ll} E \to E + T, & ET+ \\ E \to T, & T \\ T \to T * F, & TF+ \\ T \to F, & F \\ F \to id, & id \\ F \to (E), & E \end{array}$$

Найдем выход схемы для входа $id*(id+id)$. Нетрудно видеть, что существует последовательность шагов вывода

$(E, E) \Rightarrow (T, T) \Rightarrow (T * F, TF*) \Rightarrow (F * F, FF*) \Rightarrow (id * F, id\, F*) \Rightarrow (id * (E), id\, E*) \Rightarrow (id * (E + T), id\, E\, T + *) \Rightarrow (id * (T + T), id\, T\, T + *) \Rightarrow (id * (F + T), id\, F\, T + *) \Rightarrow (id * (id + T), id\, id\, T + *) \Rightarrow (id * (id + F), id\, id\, F + *) \Rightarrow (id * (id + id), id\, id\, id + *)$,
переводящая эту цепочку в цепочку $id\, id\, id + *$.

Рассмотрим связь между переводами, определяемыми СУ-схемами и осуществляемыми МП-преобразователями [2].

Теорема 5.1. *Пусть P – МП-преобразователь. Существует такая простая* СУ-*схема Tr, что $\tau(Tr) = \tau(P)$.*

Теорема 5.2. *Пусть Tr – простая* СУ-*схема. Существует такой* МП-*преобразователь* P, *что $\tau(P) = \tau(Tr)$.*

Таким образом, класс переводов, определяемых магазинными преобразователями, совпадает с классом простых СУ-переводов.

Рассмотрим теперь связь между СУ-переводами и детерминированными МП-преобразователями, выполняющими нисходящий или восходящий разбор [2].

Теорема 5.3. *Пусть $Tr = (N, T, \Pi, R, S)$ – простая* СУ-*схема, входной грамматикой которой служит* LL(1)*-грамматика. Тогда перевод $\{x\$, y) | (x, y) \in \tau(Tr)\}$ можно осуществить детерминированным* МП-*преобразователем.*

Существуют простые СУ-схемы, имеющие в качестве входных грамматик LR(1)-грамматики и не реализуемые ни на каком ДМП-преобразователе.

Пример 5.3. Рассмотрим простую СУ-схему с правилами

$$\begin{array}{ll} S \to Sa, & aSa \\ S \to Sb, & bSb \\ S \to e, & e \end{array}$$

Входная грамматика является LR(1)-грамматикой, но не существует ДМП-преобразователя, определяющего перевод $\{(x\$, y) | (x, y) \in \tau(Tr)\}$.

Назовем СУ-схему $Tr = (N, T, \Pi, R, S)$ *постфиксной*, если каждое правило из R имеет вид $A \to u, v$, где $v \in N^*\Pi^*$. Иными словами, каждый элемент перевода представляет собой цепочку из нетерминалов, за которыми следует цепочка выходных символов.

Теорема 5.4. *Пусть Tr – простая постфиксная СУ-схема, входная грамматика для которой является* LR(1). *Тогда перевод*

$$\{(x\$, y) | (x, y) \in \tau(Tr)\}$$

можно осуществить детерминированным МП-преобразователем.

5.2.2 Обобщенные схемы синтаксически управляемого перевода

Расширим определение СУ-схемы, с тем чтобы выполнять более широкий класс переводов. Во-первых, позволим иметь в каждой вершине дерева разбора несколько переводов. Как и в обычной СУ-схеме, каждый перевод зависит от прямых потомков соответствующей вершины дерева. Во-вторых, позволим элементам перевода быть произвольными цепочками выходных символов и символов, представляющих переводы в потомках. Таким образом, символы перевода могут повторяться или вообще отсутствовать.

Определение. *Обобщенной схемой синтаксически управляемого перевода* (или трансляции, сокращенно: ОСУ-схемой) называется шестерка $Tr = (N, T, \Pi, \Gamma, R, S)$, где все символы имеют тот же смысл, что и для СУ-схемы, за исключением того, что

(1) Γ – конечное множество *символов перевода* вида A_i, где $A \in N$ и i – целое число;

(2) R – конечное множество *правил перевода* вида

$A \to u, A_1 = v_1, \ldots, A_m = v_m$,

удовлетворяющих следующим условиям:

(а) $A_j \in \Gamma$ для $1 \leqslant j \leqslant m$,

(б) каждый символ, входящий в $v_1, \ldots, v_m$, либо принадлежит Π, либо является $B_k \in \Gamma$, где B входит в u,

(в) если u имеет более одного вхождения символа B, то каждый символ B_k во всех v соотнесен (верхним индексом) с конкретным вхождением B.

$A \to u$ называют *входным правилом* вывода, A_i – *переводом* нетерминала A, $A_i = v_i$ – *элементом* перевода, связанным с этим правилом перевода. Если в ОСУ-схеме нет двух правил перевода с одинаковым входным правилом вывода, то ее называют *семантически однозначной*.

Выход ОСУ-схемы определим снизу вверх. С каждой внутренней вершиной n дерева разбора (во входной грамматике), помеченной A, свяжем одну цепочку для каждого A_i. Эта цепочка называется *значением* (или переводом) символа A_i в вершине n. Каждое значение вычисляется подстановкой значений символов перевода данного элемента перевода $A_i = v_i$, определенных в прямых потомках вершины n.

Переводом $\tau(Tr)$, *определяемым* ОСУ-*схемой* Tr, назовем множество $\{(x, y) \mid x$ имеет дерево разбора во входной грамматике для Tr и y – значение выделенного символа перевода S_k в корне этого дерева$\}$.

Пример 5.4. Рассмотрим формальное дифференцирование выражений, включающих константы 0 и 1, переменную x, функции $\sin$ и $\cos$, а также операции $*$ и $+$. Такие выражения порождает грамматика

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \sin(E) \mid \cos(E) \mid x \mid 0 \mid 1$$

Свяжем с каждым из E, T и F два перевода, обозначенных индексом 1 и 2. Индекс 1 указывает на то, что выражение не дифференцировано, 2 – что выражение продифференцировано. Формальная производная – это E_2. Законы дифференцирования таковы:

$$d(f(x) + g(x)) = df(x) + dg(x)$$
$$d(f(x) * g(x)) = f(x) * dg(x) + g(x) * df(x)$$
$$d\sin(f(x)) = \cos(f(x)) * df(x)$$
$$d\cos(f(x)) = -\sin(f(x))df(x)$$
$$dx = 1$$
$$d0 = 0$$
$$d1 = 0$$

Эти законы можно реализовать следующей ОСУ-схемой:

$E \to E + T$ $\quad E_1 = E_1 + T_1$, $E_2 = E_2 + T_2$

$E \to T$ $\quad E_1 = T_1$, $E_2 = T_2$

$T \to T * F$ $\quad T_1 = T_1 * F_1$, $T_2 = T_1 * F_2 + T_2 * F_1$

$T \to F$ $\quad T_1 = F_1$, $T_2 = F_2$

$F \to (E)$ $\quad F_1 = (E_1)$, $F_2 = (E_2)$

$F \to \sin(E)$ $\quad F_1 = \sin(E_1)$, $F_2 = \cos(E_1) * (E_2)$

$F \to \cos(E)$ $\quad F_1 = \cos(E_1)$, $F_2 = -\sin(E_1) * (E_2)$

$F \to x$ $\quad F_1 = x$, $F_2 = 1$

$F \to 0$ $\quad F_1 = 0$, $F_2 = 0$

$F \to 1$ $\quad F_1 = 1$, $F_2 = 0$

Дерево вывода для $\sin(\cos(x)) + x$ приведено на рис. 5.1.

5.3 Атрибутные грамматики

Среди всех формальных методов описания языков программирования атрибутные грамматики (введенные Кнутом [6]) получили, по-видимому, наибольшую известность и распространение. Причиной этого является то, что формализм атрибутных грамматик основывается на дереве разбора программы в КС-грамматике, что сближает его с хорошо разработанной теорией и практикой построения трансляторов.

5.3.1 Определение атрибутных грамматик

Атрибутной грамматикой называется четверка $AG = (G, A_S, A_I, R)$, где

(1) $G = (N, T, P, S)$ – приведенная КС-грамматика;

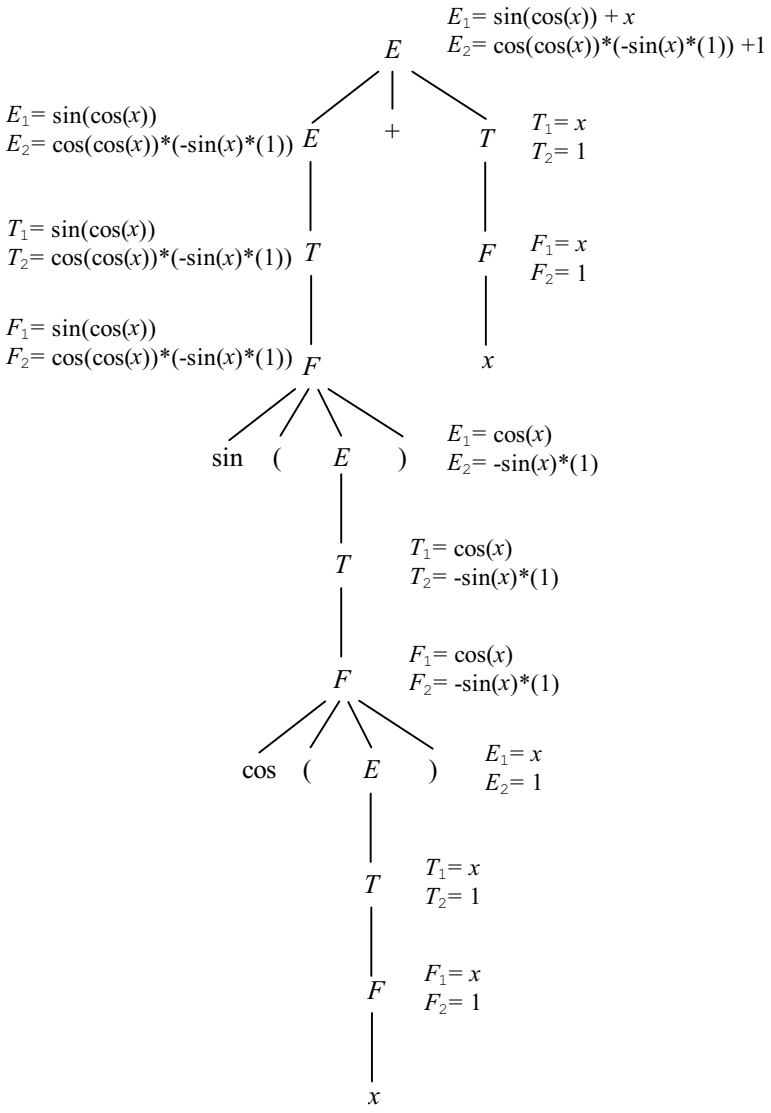

E1= sin(cos(x)) + x
E2= cos(cos(x))*(-sin(x)*(1)) +1
E
E1= sin(cos(x))
E2= cos(cos(x))*(-sin(x)*(1))
E
+
T
T1= x
T2= 1
T1= sin(cos(x))
T2= cos(cos(x))*(-sin(x)*(1))
T
F
F1= x
F2= 1
F1= sin(cos(x))
F2= cos(cos(x))*(-sin(x)*(1))
F
x
sin
(
E
)
E1= cos(x)
E2= -sin(x)*(1)
T
T1= cos(x)
T2= -sin(x)*(1)
F
F1= cos(x)
F2= -sin(x)*(1)
cos
(
E
)
E1= x
E2= 1
T
T1= x
T2= 1
F
F1= x
F2= 1
x

Рис. 5.1:

(2) A_S – конечное множество *синтезируемых атрибутов*;

(3) A_I – конечное множество *наследуемых атрибутов*, $A_S \cap A_I = \varnothing$;

(4) R – конечное множество *семантических правил*.

Атрибутная грамматика AG сопоставляет каждому символу X из $N \cup T$ множество $A_S(X)$ синтезируемых атрибутов и множество $A_I(X)$ наследуемых атрибутов. Множество всех синтезируемых атрибутов всех символов из $N \cup T$ обозначается A_S, наследуемых – A_I. Атрибуты разных символов являются различными атрибутами. Будем обозначать атрибут a символа X как $a(X)$. Значения атрибутов могут быть произвольных типов, например, представлять собой числа, строки, адреса памяти и т.д.

Пусть правило p из P имеет вид $X_0 \to X_1 X_2 \ldots X_n$. Атрибутная грамматика AG сопоставляет каждому правилу p из P конечное множество $R(p)$ семантических правил вида

$$a(X_i) = f(b(X_j), c(X_k), \ldots, d(X_m))$$

где $0 \leqslant j, k, \ldots, m \leqslant n$, причем $1 \leqslant i \leqslant n$, если $a(X_i) \in A_I(X_i)$ (т.е. $a(X_i)$ – наследуемый атрибут), и $i = 0$, если $a(X_i) \in A_S(X_i)$ (т.е. $a(X_i)$ – синтезируемый атрибут).

Таким образом, семантическое правило определяет значение атрибута a символа X_i на основе значений атрибутов b, c, ..., d символов X_j, X_k, ..., X_m соответственно.

В частном случае длина n правой части правила может быть равна нулю, тогда будем говорить, что атрибут a символа X_i "получает в качестве значения константу".

В дальнейшем будем считать, что атрибутная грамматика не содержит семантических правил для вычисления атрибутов терминальных символов. Предполагается, что атрибуты терминальных символов – либо предопределенные константы, либо доступны как результат работы лексического анализатора.

Пример 5.5. Рассмотрим атрибутную грамматику, позволяющую вычислить значение вещественного числа, представленного в десятичной записи. Здесь $N = \{Num, Int, Frac\}$, $T = \{digit, .\}$, $S = Num$, а правила вывода и семантические правила определяются следующим образом (верхние индексы используются для ссылки на разные вхождения одного и того же нетерминала):

$Num \to Int\,.\,Frac$	$v(Num) = v(Int) + v(Frac)$ $p(Frac) = 1$
$Int \to e$	$v(Int) = 0$ $p(Int) = 0$
$Int^1 \to\ digit\ Int^2$	$v(Int^1) = v(digit) * 10^{p(Int^2)} + v(Int^2)$ $p(Int^1) = p(Int^2) + 1$
$Frac \to e$	$v(Frac) = 0$
$Frac^1 \to digit\ Frac^2$	$v(Frac^1) = v(digit) * 10^{-p(Frac^1)} + v(Frac^2)$ $p(Frac^2) = p(Frac^1) + 1$

Для этой грамматики

$$A_S(Num) = \{v\}, \quad A_I(Num) = \varnothing,$$
$$A_S(Int) = \{v, p\}, \quad A_I(Int) = \varnothing,$$
$$A_S(Frac) = \{v\}, \quad A_I(Frac) = \{p\}.$$

Пусть дана атрибутная грамматика AG и цепочка, принадлежащая языку, определяемому соответствующей $G = (N, T, P, S)$. Сопоставим этой цепочке "значение" следующим образом. Построим дерево разбора T этой цепочки в грамматике G. Каждый внутренний узел этого дерева помечается нетерминалом X_0, соответствующим применению p-го правила грамматики; таким образом, у этого узла будет n непосредственных потомков (рис. 5.2).

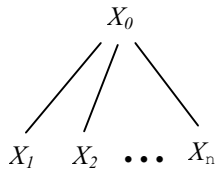


Рис. 5.2:

Пусть теперь X – метка некоторого узла дерева и пусть a – атрибут символа X. Если a – синтезируемый атрибут, то $X = X_0$ для некоторого $p \in P$; если же a – наследуемый атрибут, то $X = X_j$ для некоторых $p \in P$ и $1 \leqslant j \leqslant n$. В обоих случаях дерево "в районе" этого узла имеет вид, приведенный на рис. 5.2. По определению, атрибут a имеет в этом узле значение v, если в соответствующем семантическом правиле

$$a(X_i) = f(b(X_j), c(X_k), \ldots, d(X_m))$$

все атрибуты b, c, ..., d уже определены и имеют в узлах с метками X_j, X_k, ..., X_m значения v_j, v_k, ..., v_m соответственно, а $v = f(v_1, v_2, \ldots, v_m)$.

Процесс вычисления атрибутов на дереве продолжается до тех пор, пока нельзя будет вычислить больше ни одного атрибута. Вычисленные в результате атрибуты корня дерева представляют собой “значение”, соответствующее данному дереву вывода.

Заметим, что значение синтезируемого атрибута символа в узле синтаксического дерева вычисляется по атрибутам символов в потомках этого узла; значение наследуемого атрибута вычисляется по атрибутам “родителя’ и “соседей”.

Атрибуты, сопоставленные вхождениям символов в дерево разбора, будем называть *вхождениями* атрибутов в дерево разбора, а дерево с сопоставленными каждой вершине атрибутами – *атрибутированным деревом разбора*.

Пример 5.6. Атрибутированное дерево для грамматики из предыдущего примера и цепочки $w = 12.34$ показано на рис. 5.3.

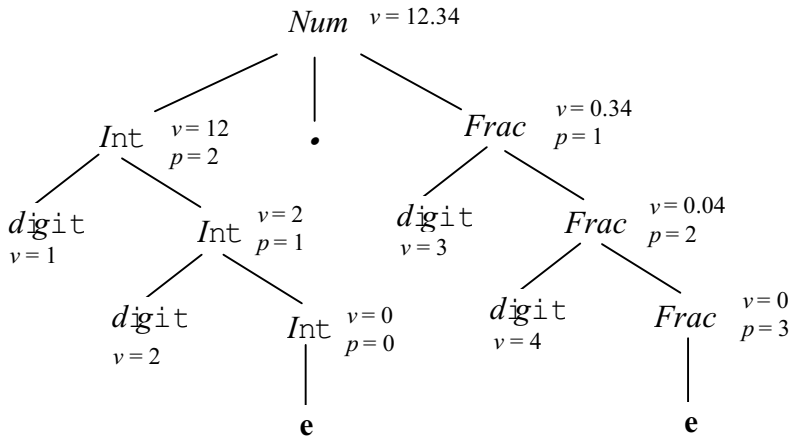


Рис. 5.3:

Будем говорить, что семантические правила *заданы корректно*, если они позволяют вычислить все атрибуты произвольного узла в любом дереве вывода.

Между вхождениями атрибутов в дерево разбора существуют зависимости, определяемые семантическими правилами, соответствующими примененным синтаксическим правилам. Эти зависимости могут быть представлены в виде ориентированного графа следующим образом.

Пусть T – дерево разбора. Сопоставим этому дереву ориентированный граф $D(T)$, узлами которого являются пары (n, a), где n – узел дерева T, a – атрибут символа, служащего меткой узла n. Граф содержит дугу из (n_1, a_1) в (n_2, a_2) тогда и только тогда, когда семантическое правило, вычисляющее атрибут a_2, непосредственно использует значение

атрибута a_1. Таким образом, узлами графа $D(T)$ являются атрибуты, которые нужно вычислить, а дуги определяют зависимости, подразумевающие, какие атрибуты вычисляются раньше, а какие позже.

Пример 5.7. Граф зависимостей атрибутов для дерева разбора из предыдущего примера показан на рис. 5.4.

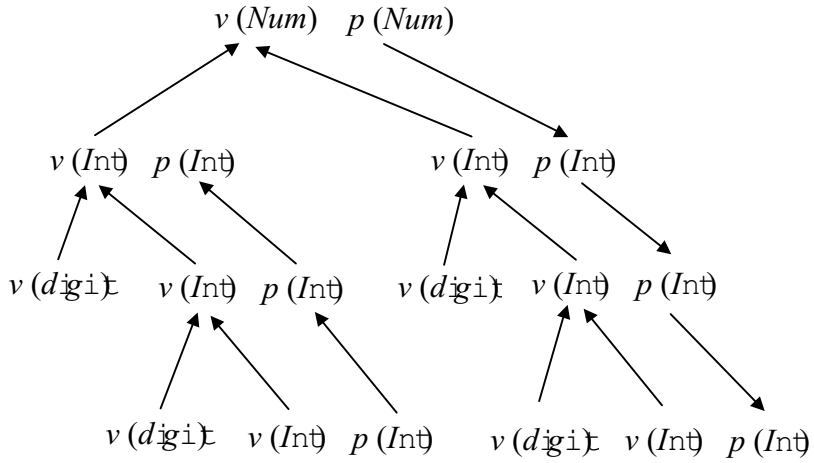


Рис. 5.4:

Можно показать, что семантические правила являются корректными тогда и только тогда, когда для любого дерева вывода T соответствующий граф $D(T)$ не содержит циклов (т.е. является ориентированным ациклическим графом).

5.3.2 Классы атрибутных грамматик и их реализация

В общем виде реализация вычислителей для атрибутных грамматик вызывает значительные трудности. Это связано с тем, что множество значений атрибутов, связанных с данным деревом, приходится вычислять в соответствии с зависимостями атрибутов, которые образуют ориентированный ациклический граф. На практике стараются осуществлять процесс вычисления атрибутов, привязывая его к тому или иному способу обхода дерева. Рассматривают многовизитные, многопроходные и другие атрибутные вычислители. Это, как правило, ведет к ограничению допустимых зависимостей между атрибутами, поддерживаемых вычислителем.

Простейшими подклассами атрибутных грамматик, вычисления всех атрибутов для которых может быть осуществлено одновременно с синтаксическим анализом, являются S-атрибутные и L-атрибутные грамматики.

Определение. Атрибутная грамматика называется S-атрибутной, если она содержит только синтезируемые атрибуты.

Нетрудно видеть, что для S-атрибутной грамматики на любом дереве разбора все атрибуты могут быть вычислены за один обход дерева снизу вверх. Таким образом, вычисление атрибутов можно делать параллельно с восходящим синтаксическим анализом, например, LR(1)-анализом.

Пример 5.8. Рассмотрим S-атрибутную грамматику для перевода арифметических выражений в ПОЛИЗ. Здесь атрибут v имеет строковый тип, $\|$ – обозначает операцию конкатенации. Правила вывода и семантические правила определяются следующим образом

$$E^1 \to E^2 + T \qquad v(E^1) = v(E^2) \,\|\, v(T) \,\|\, '+'$$

$$E \to T \qquad v(E) = v(T)$$

$$T \to T * F \qquad v(T^1) = v(T^2) \,\|\, v(F) \,\|\, '*'$$

$$T \to F \qquad v(T) = v(F)$$

$$F \to id \qquad v(F) = v(id)$$

$$F \to (E) \qquad v(F) = v(E)$$

Определение. Атрибутная грамматика называется L-атрибутной, если любой наследуемый атрибут любого символа X_j из правой части каждого правила $X_0 \to X_1 X_2 \ldots X_n$ грамматики зависит только от

(1) атрибутов символов $X_1, X_2, \ldots, X_{j-1}$, находящихся в правиле слева от X_j, и

(2) наследуемых атрибутов символа X_0.

Заметим, что каждая S-атрибутная грамматика является L-атрибутной. Все атрибуты на любом дереве для L-атрибутной грамматики могут быть вычислены за один обход дерева сверху-вниз слева-направо. Таким образом, вычисление атрибутов можно осуществлять параллельно с нисходящим синтаксическим анализом, например, LL(1)-анализом или рекурсивным спуском.

В случае рекурсивного спуска в каждой функции, соответствующей нетерминалу, надо определить формальные параметры, передаваемые по значению, для наследуемых атрибутов, и формальные параметры, передаваемые по ссылке, для синтезируемых атрибутов. В качестве примера рассмотрим реализацию атрибутной грамматики из примера 5.5 (нетрудно видеть, что грамматика является L-атрибутной).

```
void int_part(float * V0, int * P0)
{if (Map[InSym]==Digit)
  {  int I=InSym;
     float V2;
     int P2;
     InSym=getInSym();
     int_part(&V2,&P2);
     *V0=I*exp(P2*ln(10))+V2;
     *P0=P2+1;
  else {*V0=0;
        *P0=0;
     }
}
void fract_part(float * V0, int P0)
{if (Map[InSym]==Digit)
  {  int I=InSym;
     float V2;
     int P2=P0+1;
     InSym=getInSym();
     fract_part(&V2,P2);
     *V0=I*exp(-P0*ln(10))+V2;
  else {*V0=0;
     }
}
void number()
{ float V1,V3,V0;
  int P;
   int_part(&V1,&P);
   if (InSym!='.') error();
   fract_part(&V3,1);
   V0=V1+V3;
}
```

5.3.3 Язык описания атрибутных грамматик

Формализм атрибутных грамматик оказался очень удобным средством для описания семантики языков программирования. Вместе с тем выяснилось, что реализация вычислителей для атрибутных грамматик общего вида сталкивается с большими трудностями. В связи с этим было сделано множество попыток рассматривать те или иные классы атрибутных грамматик, обладающих “хорошими” свойствами. К числу таких свойств относятся прежде всего простота алгоритма проверки атрибутной грамматики на зацикленность и простота алгоритма вычисления атрибутов для атрибутных грамматик данного класса.

Атрибутные грамматики использовались для описания семантики языков программирования и было создано несколько систем автоматиза-

ции разработки трансляторов, основанных на формализме атрибутных грамматик. Опыт их использования показал, что "чистый" атрибутный формализм может быть успешно применен для описания семантики языка, но его использование вызывает трудности при создании транслятора. Эти трудности связаны как с самим формализмом, так и с некоторыми технологическими проблемами. К трудностям первого рода можно отнести несоответствие чисто функциональной природы атрибутного вычислителя и связанной с ней неупорядоченностью процесса вычисления атрибутов (что в значительной степени является преимуществом этого формализма) и упорядоченностью элементов программы. Это несоответствие ведет к тому, что приходится идти на искусственные приемы для их сочетания. Технологические трудности связаны с эффективностью трансляторов, полученных с помощью атрибутных систем. Как правило, качество таких трансляторов довольно низко из-за больших расходов памяти, неэффективности искусственных приемов, о которых было сказано выше.

Учитывая это, мы будем вести дальнейшее изложение на языке, сочетающем особенности атрибутного формализма и обычного языка программирования, в котором предполагается наличие операторов, а значит, и возможность управления порядком исполнения операторов. Этот порядок может быть привязан к обходу атрибутированного дерева разбора сверху вниз слева направо. Что касается грамматики входного языка, то мы не будем предполагать принадлежность ее определенному классу (например, LL(1) или LR(1)). Будем считать, что дерево разбора входной программы уже построено как результат синтаксического анализа и атрибутные вычисления осуществляются в результате обхода этого дерева. Таким образом, входная грамматика атрибутного вычислителя может быть даже неоднозначной, что не влияет на процесс атрибутных вычислений.

При записи синтаксиса мы будем использовать расширенную БНФ. Элемент правой части синтаксического правила, заключенный в скобки [ ], может отсутствовать. Элемент правой части синтаксического правила, заключенный в скобки ( ), означает возможность повторения один или более раз. Элемент правой части синтаксического правила, заключенный в скобки [()], означает возможность повторения ноль или более раз. В скобках [ ] или [()] может указываться разделитель конструкций.

Ниже дан синтаксис языка описания атрибутных грамматик. Приведен только синтаксис конструкций, собственно описывающих атрибутные вычисления. Синтаксис обычных выражений и операторов не приводится – он основывается на Си.

```
Атрибутная грамматика ::= 'ALPHABET'
     ( ОписаниеНетерминала ) ( Правило )
ОписаниеНетерминала ::= ИмяНетерминала
          '::' [( ОписаниеАтрибутов / ';')] '.'
ОписаниеАтрибутов ::= Тип ( ИмяАтрибута / ',')
```

```
Правило ::= 'RULE' Синтаксис 'SEMANTICS' Семантика '.'
Синтаксис ::= ИмяНетерминала '::=' ПраваяЧасть
ПраваяЧасть ::= [( ЭлементПравойЧасти )]
ЭлементПравойЧасти ::= ИмяНетерминала
      | Терминал
      | '(' Нетерминал [ '/' Терминал ] ')'
      | '[' Нетерминал ']'
      | '[(' Нетерминал [ '/' Терминал ] ')]'
Семантика ::= [(ЛокальноеОбъявление / ';')]
         [( СемантическоеДействие / ';')]
СемантическоеДействие ::= Присваивание
       | [ Метка ] Оператор
Присваивание ::= Переменная ':=' Выражение
Переменная ::= ЛокальнаяПеременная
       |  Атрибут
Атрибут ::= ЛокальныйАтрибут
     |  ГлобальныйАтрибут

ЛокальныйАтрибут ::= ИмяАтрибута '<' Номер '>'
ГлобальныйАтрибут ::= ИмяАтрибута '<' Нетерминал '>'
Метка ::= Целое ':'
    |  Целое 'E' ':'
    |  Целое 'A' ':'
Оператор ::= Условный
      |  ОператорПроцедуры
      |  ЦиклПоМножеству
      |  ПростойЦикл
      |  ЦиклСУсловиемОкончания
```

Описание атрибутной грамматики состоит из раздела описания атрибутов и раздела правил. Раздел описания атрибутов определяет состав атрибутов для каждого символа грамматики и тип каждого атрибута. Правила состоят из синтаксической и семантической части. В синтаксической части используется расширенная БНФ. Семантическая часть правила состоит из локальных объявлений и семантических действий. В качестве семантических действий допускаются как атрибутные присваивания, так и составные операторы.

Метка в семантической части правила привязывает выполнение оператора к обходу дерева разбора сверху-вниз слева направо. Конструкция i : оператор означает, что оператор должен быть выполнен сразу после обхода i-й компоненты правой части. Конструкция i E : оператор означает, что оператор должен быть выполнен, только если порождение i-й компоненты правой части пусто. Конструкция i A : оператор означает, что оператор должен быть выполнен после разбора каждого повторения i-й компоненты правой части (имеется в виду конструкция повторения).

Каждое правило может иметь локальные определения (типов и пе-

ременных). В формулах используются как атрибуты символов данного правила (*локальные* атрибуты) и в этом случае соответствующие символы указываются номерами в правиле (0 – для символа левой части, 1 – для первого символа правой части, 2 – для второго символа правой части и т.д.), так и атрибуты символов предков левой части правила (*глобальные* атрибуты). В этом случае соответствующий символ указывается именем нетерминала. Таким образом, на дереве образуются области видимости атрибутов: атрибут символа имеет область видимости, состоящую из правила, в которое символ входит в правую часть, плюс все поддерево, корнем которого является символ, за исключением поддеревьев – потомков того же символа в этом поддереве.

Значение терминального символа доступно через атрибут VAL соответствующего типа.

Пример 5.9. Атрибутная грамматика из примера 5.5 записывается следующим образом:

```
ALPHABET
Num   :: float V.
Int   :: float V;
         int P.
Frac  :: float V;
         int P.
digit :: int VAL.

RULE
Num ::= Int '.' Frac
SEMANTICS
V<0>=V<1>+V<3>; P<3>=1.

RULE
Int ::= e
SEMANTICS
V<0>=0; P<0>=0.

RULE
Int ::= digit Int
SEMANTICS
V<0>=VAL<1>*10**P<2>+V<2>; P<0>=P<2>+1.

RULE
Frac ::= e
SEMANTICS
V<0>=0.

RULE
Frac ::= digit Frac
SEMANTICS
V<0>=VAL<1>*10**(-P<0>)+V<2>; P<2>=P<0>+1.
```

Глава 6

Проверка контекстных условий

6.1 Описание областей видимости и блочной структуры

Задачей контекстного анализа является установление свойств объектов и их использования. Наиболее часто решаемой задачей является определение существования объекта и соответствия его использования контексту, что осуществляется с помощью анализа типа объекта. Под контекстом здесь понимается вся совокупность свойств текущей точки программы, например множество доступных объектов, тип выражения и т.д.

Таким образом, необходимо хранить объекты и их типы, уметь находить эти объекты и определять их типы, определять характеристики контекста. Совокупность доступных в данной точке объектов будем называть *средой*. Обычно среда программы состоит из частично упорядоченного набора компонент

$$E = \{DS_1, DS_2, \dots, DS_n\}$$

Каждая компонента – это множество объявлений, представляющих собой пары (имя, тип):

$$DS_i = \{(\text{имя}_j, \text{тип}_j) \mid 1 \leqslant j \leqslant k_i\}$$

где под типом будем подразумевать полное описание свойств объекта (объектом, в частности, может быть само описание типа).

Компоненты образуют дерево, соответствующее этому частичному порядку. Частичный порядок между компонентами обычно определяется статической вложенностью компонент в программе. Эта вложенность может соответствовать блокам, процедурам или классам программы (рис. 6.1).

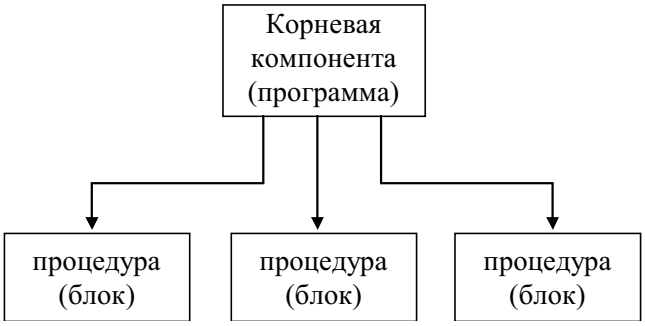


Рис. 6.1:

Компоненты среды могут быть именованы. Поиск в среде обычно ведется с учетом упорядоченности компонент. Среда может включать в себя как компоненты, полученные при трансляции "текущего" текста программы, так и "внешние" (например, раздельно компилированные) компоненты.

Для обозначения участков программы, в которых доступны те или иные описания, используются понятия *области действия* и *области видимости*. Областью действия описания является процедура (блок), содержащая описание, со всеми входящими в нее (подчиненными по дереву) процедурами (блоками). Областью видимости описания называется часть области действия, из которой исключены те подобласти, в которых по тем или иным причинам описание недоступно, например, оно перекрыто другим описанием. В разных языках понятия области действия и области видимости уточняются по-разному.

Обычными операциями при работе со средой являются:

– включить объект в компоненту среды;

– найти объект в среде и получить доступ к его описанию;

– образовать в среде новую компоненту, определенным образом связанную с остальными;

– удалить компоненту из среды.

Среда состоит из отдельных объектов, реализуемых как записи (в дальнейшем описании мы будем использовать имя TElement для имени типа этой записи). Состав полей записи, вообще говоря, зависит от описываемого объекта (тип, переменная и т.д.), но есть поля, входящие в запись для любого объекта:

TObject Object – категория объекта (тип, переменная, процедура и т.д.);

TMode Mode – вид объекта: целый, массив, запись и т.д.;

TName Name – имя объекта;

TType Type – указатель на описание типа.

6.2 Занесение в среду и поиск объектов

Рассмотрим схему реализации простой блочной структуры, аналогичной процедурам в Паскале или блокам в Си. Каждый блок может иметь свой набор описаний. Программа состоит из основного именованного блока, в котором имеются описания и операторы. Описания состоят из описаний типов и объявлений переменных. В качестве типа может использоваться целочисленный тип и тип массива. Два типа T1 и T2 считаются эквивалентными, если имеется описание T1=T2 (или T2=T1). Операторами служат операторы присваивания вида Переменная1=Переменная2 и блоки. Переменная – это либо просто идентификатор, либо выборка из массива. Оператор присваивания считается правильным, если типы переменных левой и правой части эквивалентны. Примером правильной программы может служить

```
program Example
begin
 type  T1=array 100 of array 200 of integer;
     T2=T1;
 var   V1:T1;
     V2:T2;
 begin
  V1=V2;
  V2[1]=V1[2];
  begin
   type  T3=array 300 of T1;
   var   V3:T3;
   V3[50]=V1;
  end
 end
end.
```

Рассматриваемое подмножество языка может быть порождено следующей грамматикой (запись в расширенной БНФ):

```
Prog ::= 'program' Ident Block '.'
Block ::= 'begin' [( Declaration )] [ (Statement) ] 'end'
Declaration ::= 'type' ( Type_Decl )
Type_Decl ::= Ident '=' Type_Defin
Type_Defin ::= 'ARRAY' Index 'OF' Type_Defin
Type_Defin ::= Type_Use
Type_Use ::= Ident
Declaration ::= 'var' ( Var_Decl )
Var_Decl ::= Ident_List ':' Type_Use ';'
Ident_List ::= ( Ident / ',' )
Statement ::= Block ';'
Statement ::= Variable '=' Variable ';'
```

```
Variable ::= Ident Access
Access ::= '[' Expression ']' Access
Access ::=
```

Для реализации некоторых атрибутов (в частности среды, списка идентификаторов и т.д.) в качестве типов данных мы будем использовать различные множества. Множество может быть упорядоченным или неупорядоченным, ключевым или простым. Элементом ключевого множества может быть запись, одним из полей которой является ключ:

SETOF T – простое неупорядоченное множество объектов типа T;

KEY K SETOF T – ключевое неупорядоченное множество объектов типа T с ключом типа K;

LISTOF T – простое упорядоченное множество объектов типа T;

KEY K LISTOF T – ключевое упорядоченное множество объектов типа T с ключом типа K;

Над объектами типа множества определены следующие операции:

Init(S) – создать и проинициализировать переменную S;

Include(V,S) – включить объект V в множество S; если множество упорядоченное, то включение осуществляется в качестве последнего элемента;

Find(K,S) – выдать указатель на объект с ключом K во множестве S и NIL, если объект с таким ключом не найден.

Имеется специальный оператор цикла, пробегающий элементы множества:

```
for (V in S) Оператор;
```

Переменная V пробегает все значения множества. Если множество упорядочено, то элементы пробегаются в этом порядке, если нет – в произвольном порядке.

Среда представляет собой ключевое множество с ключом – именем объекта. Идентификаторы имеют тип TName. Обозначение <Нетерминал> в позиции типа – это указатель на вершину типа Нетерминал. Обозначение <Нетерминал> в выражении – это взятие значения указателя на ближайшую вершину вверх по дереву разбора, помеченную соответствующим нетерминалом.

Для реализации среды каждый нетерминал Block имеет атрибут Env. Для обеспечения возможности просматривать компоненты среды в соответствии с вложенностью блоков каждый нетерминал Block имеет атрибут Pred – указатель на охватывающий блок. Кроме того, среда блока

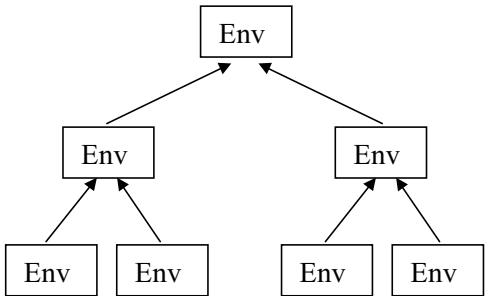


Рис. 6.2:

корня дерева (нетерминал Prog) содержит все предопределенные описания (рис. 6.2). Это заполнение реализуется процедурой PreDefine. Атрибут Pred блока корневой компоненты имеет значение NULL.

Атрибутная реализация выглядит следующим образом.

```
// Описание атрибутов
ALPHABET

Prog:: KEY TName SETOF TElement Env.
// Корневая компонента, содержащая предопределенные описания.

Block:: KEY TName SETOF TElement Env;
      <Block> Pred.

Ident_List:: SETOF TName Ident_Set.
// Ident_Set - список идентификаторов

Type_Defin, Type_Use, Access, Expression:: TType ElementType.
// ElementType - указатель на описание типа

Declaration, Var_Decl, Type_Decl::.

Ident:: TName Val.

Index:: int Val.

// Описание синтаксисических и семантических правил
RULE
Prog ::= 'program' Ident Block '.'
SEMANTICS
0:{Init(Env<3>);
  PreDefine(Env<3>);
```

```
  Pred<3>=NULL
 }.

RULE
Block ::= 'begin' [( Declaration )] [ (Statement) ] 'end'
SEMANTICS
0: if (<Block>!=NULL){
    Init(Env<0>);
    Pred<0>=<Block>
  }.

RULE
Declaration ::= 'type' ( Type_Decl ).

RULE
Type_Decl ::= Ident '=' Type_Defin
SEMANTICS
TElement V;
if (Find(Val<1>,Env<Block>)!=NULL)
   Error("Identifier declared twice");
// Идентификатор уже объявлен в блоке
// В любом случае заносится новое описание
V.Name=Val<1>;
V.Object=TypeObject;
V.Type=ElementType<3>;
Include(V,Env<Block>).

RULE
Type_Defin ::= 'ARRAY' Index 'OF' Type_Defin
SEMANTICS
ElementType<0>=ArrayType(ElementType<4>,Val<2>).

RULE
Type_Defin ::= Type_Use
SEMANTICS
ElementType<0>=ElementType<1>.

RULE
Type_Use ::= Ident
SEMANTICS
TElement * PV;
PV=FindObject(Val<1>,<Block>,TypeObject,<Prog>);
ElementType<0>=PV->Type.
// В этом правиле анализируется использующая позиция
// идентификатора типа.
```

```
RULE
Declaration ::= 'var' ( Var_Decl ).

RULE
Var_Decl ::= Ident_List ':' Type_Use ';'
SEMANTICS
TElement V;
TName N;
for (N in Ident_Set<1>){
// Цикл по (неупорядоченному) списку идентификаторов
  if (Find(N,Env<Block>)!=NULL)
     Error("Identifier declared twice");
// Идентификатор уже объявлен в блоке
// В любом случае заносится новое описание
  V.Name=N;
  V.Object=VarObject;
  V.Type=ElementType<3>;
  Include(V,Env<Block>)
}.
// N - рабочая переменная для элементов списка. Для каждого
// идентификатора из множества идентификаторов Ident_Set<1>
// сформировать объект-переменную в текущей компоненте среды
// с соответствующими характеристиками.

RULE
Ident_List ::= ( Ident /',' )
SEMANTICS
0:Init(Ident_Set<0>);
1A:Include(Val<1>,Ident_Set<0>).

RULE
Statement ::= Block ';'.

RULE
Statement ::= Variable '=' Variable ';'
SEMANTICS
if (ElementType<1>!=NULL) && (ElementType<3>!=NULL)
       && (ElementType<1>!=ElementType<3>)
   Error("Incompatible Expression Types").

RULE
Variable ::= Ident Access
SEMANTICS
TElement * PV;
PV=FindObject(Val<1>,<Block>,VarObject,<Prog>);
if (PV==NULL){
```

```
  Error("Identifier used is not declared");
  ElementType<2>=NULL
}
else
  ElementType<2>=PV->Type.

RULE
Access ::= '[' Expression ']' Access
SEMANTICS
ElementType<4>=ArrayElementType(ElementType<0>, ElementType<2>).

RULE
Access ::=
SEMANTICS
ElementType<Variable>=ElementType<0>.
```

Поиск в среде осуществляется следующей функцией:

```
TElement * FindObject(TName Ident, <Block> BlockPointer,
                TObject Object, <Prog> Prog)
{ TElement * ElementPointer;
// Получить указатель на ближайший охватывающий блок
do{
 ElementPointer=Find(Ident, BlockPointer->Env);
 BlockPointer=BlockPointer->Pred;
 }
while (ElementPointer==NULL)&&(BlockPointer!=NULL);
// Искать до момента, когда либо найдем нужный идентификатор,
// либо дойдем до корневой компоненты
if (ElementPointer==NULL)&&(BlockPointer==NULL)
// Дошли до корневой компоненты и еще не нашли идентификатора
// Найти объект среди предопределенных
  ElementPointer=Find(Ident, Prog->Env);
if (ElementPointer!=NULL)
// Нашли объект с данным идентификатором
// либо в очередном блоке, либо среди предопределенных
if (ElementPointer->Object!=Object){
// Проверить, имеет ли найденный объект
// нужную категорию
  Error("Object of specified category is not found");
  ElementPointer=NULL;
}
else
// Объект не найден
  Error("Object is not found");
return ElementPointer;
}
```

Переменная BlockPointer – указатель на ближайший охватывающий блок. Переходя от блока к блоку, ищем объект в его среде. Если не нашли, то переходим к охватывающему блоку. Если дошли до корневой компоненты, пытаемся найти объект среди предопределенных объектов. Если объект нашли, надо убедиться, что он имеет нужную категорию.

Функция ArrayElementType(TType EntryType, TType ExprType) осуществляет проверку допустимости применения операции взятия индекса к переменной и возвращает тип элемента массива.

Функция ArrayType(TType EntryType, int Val) возвращает описание типа – массива с типом элемента EntryType и диапазоном индекса Val.

Глава 7

Организация таблиц символов

В процессе работы компилятор хранит информацию об объектах программы в специальных таблицах символов. Как правило, информация о каждом объекте состоит из двух основных элементов: имени объекта и описания объекта. Информация об объектах программы должна быть организована таким образом, чтобы поиск ее был по возможности быстрее, а требуемая память по возможности меньше.

Кроме того, со стороны языка программирования могут быть дополнительные требования к организации информации. Имена могут иметь определенную область видимости. Например, поле записи должно быть уникально в пределах структуры (или уровня структуры), но может совпадать с именем объекта вне записи (или другого уровня записи). В то же время имя поля может открываться оператором присоединения, и тогда может возникнуть конфликт имен (или неоднозначность в трактовке имени). Если язык имеет блочную структуру, то необходимо обеспечить такой способ хранения информации, чтобы, во-первых, поддерживать блочный механизм видимости, а во-вторых – эффективно освобождать память при выходе из блока. В некоторых языках (например, Аде) одновременно (в одном блоке) могут быть видимы несколько объектов с одним именем, в других такая ситуация недопустима.

Мы рассмотрим некоторые основные способы организации таблиц символов в компиляторе: таблицы идентификаторов, таблицы расстановки, двоичные деревья и реализацию блочной структуры.

7.1 Таблицы идентификаторов

Как уже было сказано, информацию об объекте обычно можно разделить на две части: имя (идентификатор) и описание. Если длина идентификатора ограничена (или имя идентифицируется по ограниченному

числу первых символов идентификатора), то таблица символов может быть организована в виде простого массива строк фиксированной длины, как это изображено на рис. 7.1. Некоторые входы могут быть заняты, некоторые – свободны.

Рис. 7.1:

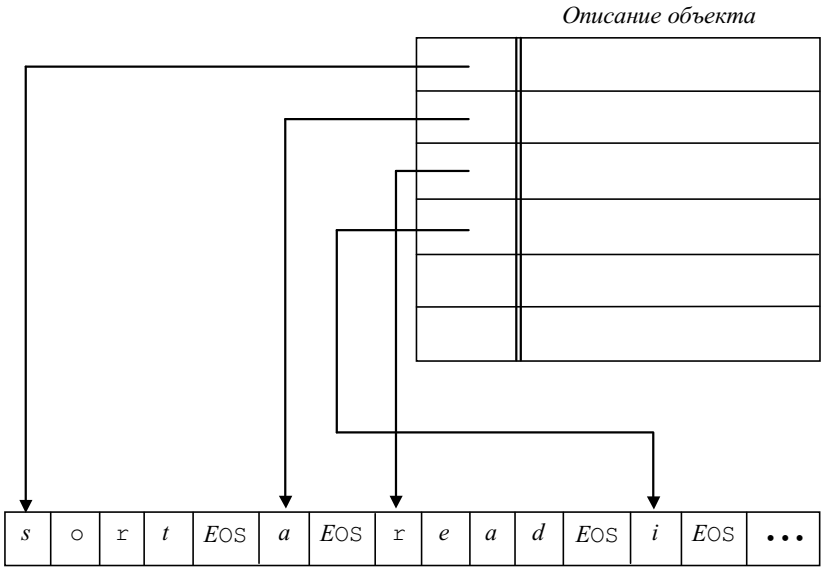


Рис. 7.2:

Ясно, что, во-первых, размер массива должен быть не меньше числа идентификаторов, которые могут реально появиться в программе (в противном случае возникает переполнение таблицы); во-вторых, как правило, потенциальное число различных идентификаторов существенно

больше размера таблицы.

Заметим, что в большинстве языков программирования символьное представление идентификатора может иметь произвольную длину. Кроме того, различные объекты в одной или в разных областях видимости могут иметь одинаковые имена, и нет большого смысла занимать память для повторного хранения идентификатора. Таким образом, удобно имя объекта и его описание хранить по отдельности.

В этом случае идентификаторы хранятся в отдельной таблице – *таблице идентификаторов*. В таблице символов же хранится указатель на соответствующий вход в таблицу идентификаторов. Таблицу идентификаторов можно организовать, например, в виде сплошного массива. Идентификатор в массиве заканчивается каким-либо специальным символом EOS (рис. 7.2). Второй возможный вариант – в качестве первого символа идентификатора в массив заносится его длина.

7.2 Таблицы расстановки

Одним из эффективных способов организации таблицы символов является *таблица расстановки* (или *хеш-таблица*). Поиск в такой таблице может быть организован методом повторной расстановки. Суть его заключается в следующем.

Таблица символов представляет собой массив фиксированного размера N. Идентификаторы могут храниться как в самой таблице символов, так и в отдельной таблице идентификаторов.

Определим некоторую функцию h_1 (*первичную функцию расстановки*), определенную на множестве идентификаторов и принимающую значения от 0 до $N-1$ (т.е. $0 \leqslant h_1(id) \leqslant N-1$, где id – символьное представление идентификатора). Таким образом, функция расстановки сопоставляет идентификатору некоторый адрес в таблице символов.

Пусть мы хотим найти в таблице идентификатор id. Если элемент таблицы с номером $h_1(id)$ не заполнен, то это означает, что идентификатора в таблице нет. Если же занят, то это еще не означает, что идентификатор id в таблицу занесен, поскольку (вообще говоря) много идентификаторов могут иметь одно и то же значение функции расстановки. Для того чтобы определить, нашли ли мы нужный идентификатор, сравниваем id с элементом таблицы $h_1(id)$. Если они равны – идентификатор найден, если нет – надо продолжать поиск дальше.

Для этого вычисляется *вторичная функция расстановки* $h_2(h)$ (значением которой опять таки является некоторый адрес в таблице символов). Возможны четыре варианта:

– элемент таблицы не заполнен (т.е. идентификатора в таблице нет),

– идентификатор элемента таблицы совпадает с искомым (т.е. идентификатор найден),

– адрес элемента совпадает с уже просмотренным (т.е. таблица вся просмотрена и идентификатора нет)

– предыдущие варианты не выполняются, так что необходимо продолжать поиск.

Для продолжения поиска применяется следующая функция расстановки $h_3(h_2)$, $h_4(h_3)$ и т.д. Как правило, $h_i = h_2$ для $i \geqslant 2$. Аргументом функции h_2 является целое в диапазоне $[0, N-1]$ и она может быть быть устроена по-разному. Приведем три варианта.

1) $h_2(i) = (i+1) \bmod N$.

Берется следующий (циклически) элемент массива. Этот вариант плох тем, что занятые элементы "группируются", образуют последовательные занятые участки и в пределах этого участка поиск становится по-существу линейным.

2) $h_2(i) = (i+k) \bmod N$, где k и N взаимно просты.

По-существу это предыдущий вариант, но элементы накапливаются не в последовательных элементах, а "разносятся".

3) $h_2(i) = (a * i + c) \bmod N$ – "псевдослучайная последовательность".

Здесь c и N должны быть взаимно просты, $b = a - 1$ кратно p для любого простого p, являщегося делителем N, b кратно 4, если N кратно 4 [5].

Поиск в таблице расстановки можно описать следующей функцией:

```
void Search(String Id,boolean * Yes,int * Point)
{int H0=h1(Id), H=H0;
  while (1)
  {if (Empty(H)==NULL)
    {*Yes=false;
     *Point=H;
     return;
    }
   else if (IdComp(H,Id)==0)
    {*Yes=true;
     *Point=H;
     return;
    }
   else H=h2(H);
   if (H==H0)
    {*Yes=false;
     *Point=NULL;
     return;
    }
  }
}
```

Функция IdComp(H,Id) сравнивает элемент таблицы на входе H с идентификатором и вырабатывает 0, если они равны. Функция Empty(H) вырабатывает NULL, если вход H пуст. Функция Search присваивает параметрам Yes и Pointer соответственно следующие значения :

true, P – если нашли требуемый идентификатор, где P – указатель на соответствующий этому идентификатору вход в таблице,

false, NULL – если искомый идентификатор не найден, причем в таблице нет свободного места, и

false, P – если искомый идентификатор не найден, но в таблице есть свободный вход P.

Занесение элемента в таблицу можно осуществить следующей функцией:

```
int Insert(String Id)
{boolean Yes;
 int Point=-1;
 Search(Id,&Yes,&Point);
 if (!Yes && (Point!=NULL)) InsertId(Point,Id);
 return(Point);
}
```

Здесь функция InsertId(Point,Id) заносит идентификатор Id для входа Point таблицы.

7.3 Таблицы расстановки со списками

Только что описанная схема страдает одним недостатком – возможностью переполнения таблицы. Рассмотрим ее модификацию, когда все элементы, имеющие одинаковое значения (первичной) функции расстановки, связываются в список (при этом отпадает необходимость использования функций h_i для $i \geqslant 2$). Таблица расстановки со списками – это массив указателей на списки элементов (рис. 7.3).

Вначале таблица расстановки пуста (все элементы имеют значение NULL). При поиске идентификатора Id вычисляется функция расстановки h(Id) и просматривается соответствующий линейный список. Поиск в таблице может быть описан следующей функцией:

```
struct Element
  {String IdentP;
   struct Element * Next;
  };
struct Element * T[N];

struct Element * Search(String Id)
{struct Element * P;
 P=T[h(Id)];
 while (1)
 {if (P==NULL) return(NULL);
  else if (IdComp(P->IdentP,Id)==0) return(P);
  else P=P->Next;
```

```
}
}
```

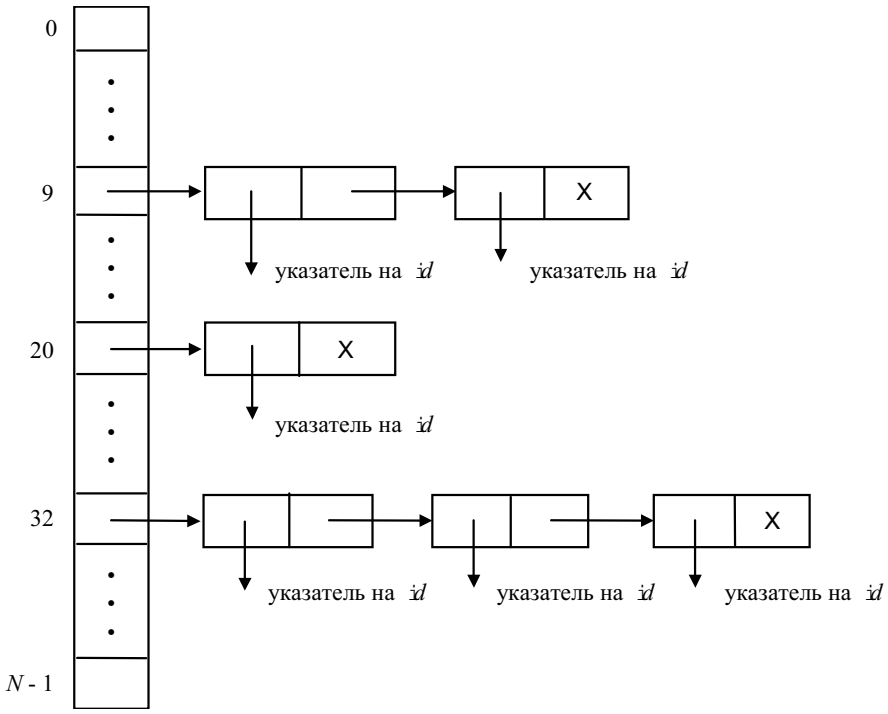

0
9
20
32
N - 1
X
X
X
X
указатель на id
указатель на id
указатель на id
указатель на id
указатель на id

Рис. 7.3:

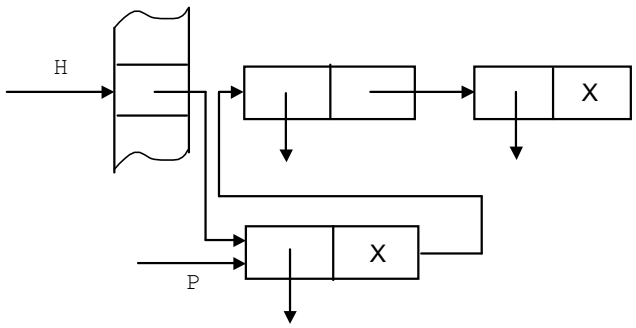

H
X
X
P

Рис. 7.4:

Занесение элемента в таблицу можно осуществить следующей функцией:

```
struct Element * Insert(String Id)
{struct Element * P,H;
 P=Search(Id);
 if (P!=NULL) return(P);
 else {H=H(Id);
      P=alloc(sizeof(struct Element));
      P->Next=T[H];
      T[H]=P;
      P->IdentP=Include(Id);
     }
 return(P);
}
```

Процедура Include заносит идентификатор в таблицу идентификаторов. Алгоритм иллюстрируется рис. 7.4.

7.4 Функции расстановки

Много внимания исследователями было уделено тому, какой должна быть (первичная) функция расстановки. Основные требования к ней очевидны: она должна легко вычисляться и распределять равномерно. Один из возможных подходов здесь заключается в следующем.

1. По символам строки s определяем положительное целое H. Преобразование одиночных символов в целые обычно можно сделать средствами языка реализации. В Паскале для этого служит функция ord, в Си при выполнении арифметических операций символьные значения трактуются как целые.

2. Преобразуем H, вычисленное выше, в номер элемента, т.е. целое между 0 и $N-1$, где N – размер таблицы расстановки, например, взятием остатка при делении H на N.

Функции расстановки, учитывающие все символы строки, распределяют лучше, чем функции, учитывающие только несколько символов, например, в конце или середине строки. Но такие функции требуют больше вычислений.

Простейший способ вычисления H – сложение кодов символов. Перед сложением с очередным символом можно умножить старое значение H на константу q. Т.е. полагаем $H_0 = 0$, $H_i = q^* H_{i-1} + c_i$ для $1 \leqslant i \leqslant k$, k – длина строки. При $q = 1$ получаем простое сложение символов. Вместо сложения можно выполнять сложение c_i и $q * H_{j-1}$ по модулю 2. Переполнение при выполнении арифметических операций можно игнорировать.

Функция Hashpjw, приведенная ниже [10], вычисляется, начиная с $H = 0$ (предполагается, что используются 32-битовые целые). Для каждого символа c сдвигаем биты H на 4 позиции влево и добавляем очередной символ. Если какой-нибудь из четырех старших бит H равен 1, сдвигаем эти 4 бита на 24 разряда вправо, затем складываем по модулю 2 с H и устанавливаем в 0 каждый из четырех старших бит, равных 1.

```
#define PRIME 211
#define EOS '\0'
int Hashpjw(char *s)
{char *p;
 unsigned H=0, g;
 for (p=s; *p!=EOS; p=p+1)
  {H=(H<<4)+(*p);
   if (g=H&0xf0000000)
     {H=H^(g>>24);
      H=H^g;
  }  }
 return H%PRIME;
}
```

7.5 Таблицы на деревьях

Рассмотрим еще один способ организации таблиц символов с использованием *двоичных деревьев*.

Ориентированное дерево называется *двоичным*, если у него в каждую вершину, кроме одной (*корня*), входит одна дуга, и из каждой вершины выходит не более двух дуг. *Ветвью* дерева называется поддерево, состоящее из некоторой дуги данного дерева, ее начальной и конечной вершин, а также всех вершин и дуг, лежащих на всех путях, выходящих из конечной вершины этой дуги. *Высотой* дерева называется максимальная длина пути в этом дереве от корня до листа.

Пусть на множестве идентификаторов задан некоторый линейный (например, лексикографический) порядок $\prec$, т.е. некоторое транзитивное, антисимметричное и антирефлексивное отношение. Таким образом, для произвольной пары идентификаторов id_1 и id_2 либо $id_1 \prec id_2$, либо $id_2 \prec id_1$, либо id_1 совпадает с id_2.

Каждой вершине двоичного дерева, представляющего таблицу символов, сопоставим идентификатор. При этом, если вершина (которой сопоставлен id) имеет левого потомка (которому сопоставлен id_L), то $id_L \prec id$; если имеет правого потомка (id_R), то $id \prec id_R$. Элемент таблицы изображен на рис. 7.5.

Поиск в такой таблице может быть описан следующей функцией:

```
struct TreeElement * SearchTree(String Id, struct TreeElement * TP)
```

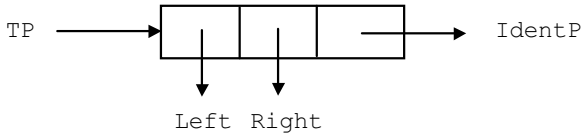


Рис. 7.5:

```
{int comp;
 if (TP==NULL) return NULL;
 comp=IdComp(Id,TP->IdentP);
 if (comp<0) return(SearchTree(Id,TP->Left));
 if (comp>0) return(SearchTree(Id,TP->Right));
 return TP;
}
```

где структура для для элемента дерева имеет вид

```
struct TreeElement
  {String IdentP;
   struct TreeElement * Left, * Right;
  };
```

Занесение в таблицу осуществляется функцией

```
struct TreeElement * InsertTree(String Id, struct TreeElement * TP)
{int comp=IdComp(Id,TP->IdentP);
 if (comp<0) return(Fill(Id,TP->Left, &(TP->Left)));
 if (comp>0) return(Fill(Id,TP->Right, &(TP->Right)));
 return(TP);
}

struct TreeElement * Fill(String Id,
                  struct TreeElement * P,
                  struct TreeElement ** FP)
{ if (P==NULL)
  {P=alloc(sizeof(struct TreeElement));
   P->IdentP=Include(Id);
   P->Left=NULL;
   P->Right=NULL;
   *FP=P;
   return(P);
  }
 else return(InsertTree(Id,P));
}
```

Как показано в работе [8], среднее время поиска в таблице размера n, организованной в виде двоичного дерева, при равной вероятности появления каждого объекта равно $(2 \ln 2) \log_2 n + O(1)$. Однако, на практике случай равной вероятности появления объектов встречается довольно редко. Поэтому в дереве появляются более длинные и более короткие ветви, и среднее время поиска увеличивается.

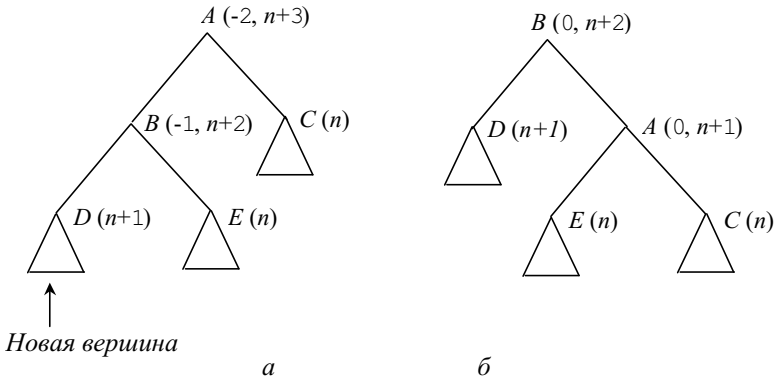


Рис. 7.6:

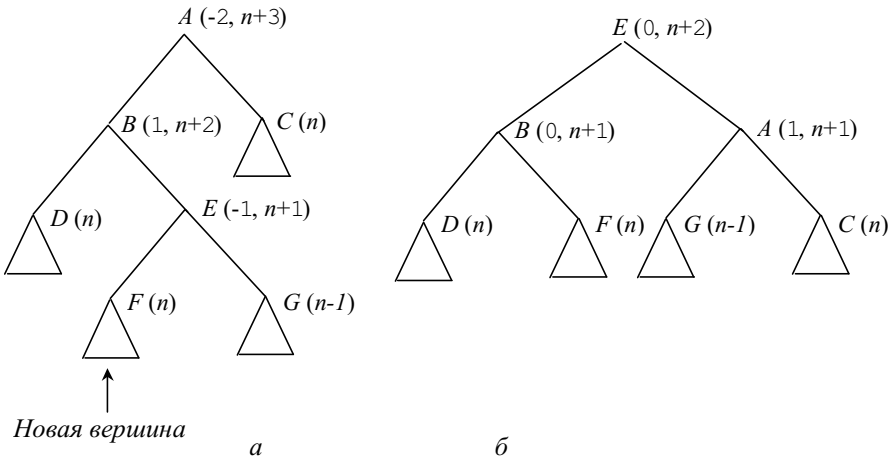
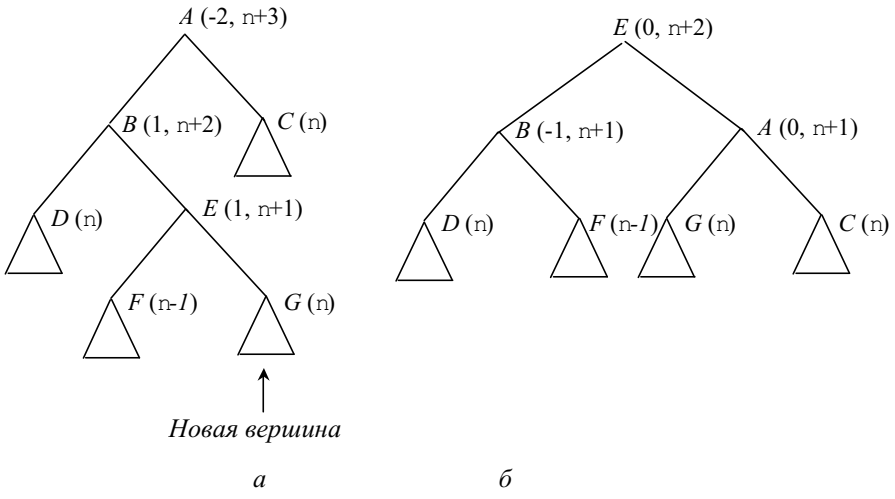


Рис. 7.7:

Чтобы уменьшить среднее время поиска в двоичном дереве, можно в процессе построения дерева следить за тем, чтобы оно все время оставалось сбалансированным. А именно, назовем дерево *сбалансированным*, если ни для какой вершины высота выходящей из нее правой ветви не отличается от высоты левой более чем на 1. Для того, чтобы достичь сба-

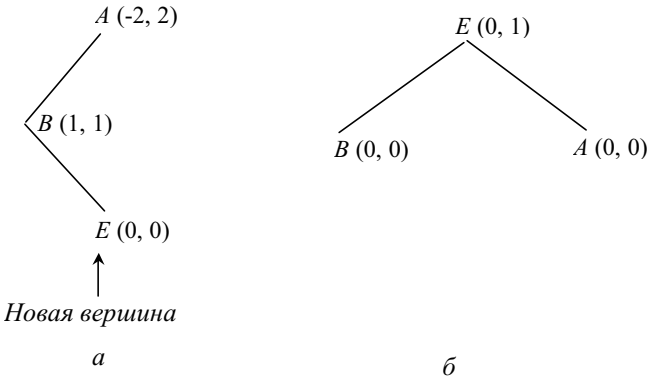
лансированности, в процессе добавления новых вершин дерево можно слегка перестраивать следующим образом [1].

Определим для каждой вершины дерева характеристику, равную разности высот выходящих из нее правой и левой ветвей. В сбалансированном дереве характеристика вершины может быть равной -1, 0 и 1, для листьев она равна 0.



а *б*

Рис. 7.8:



а *б*

Рис. 7.9:

Пусть мы определили место новой вершины в дереве. Ее характеристика равна 0. Назовем путь, ведущий от корня к новой вершине, *выделенным*. При добавлении новой вершины могут измениться характери-

стики только тех вершин, которые лежат на выделенном пути. Рассмотрим *заключительный* отрезок выделенного пути, такой, что до добавления вершины характеристики всех вершин на нем были равны 0. Если верхним концом этого отрезка является сам корень, то дерево перестраивать не надо, достаточно лишь изменить характеристики вершин на этом пути на **1** или -1, в зависимости от того, влево или вправо пристроена новая вершина.

Пусть верхний конец заключительного отрезка – не корень. Рассмотрим вершину A – "родителя" верхнего конца заключительного отрезка. Перед добавлением новой вершины характеристика A была равна ± 1. Если A имела характеристику 1 (-1) и новая вершина добавляется в левую (правую) ветвь, то характеристика вершины A становится равной 0, а высота поддерева с корнем в A не меняется. Так что и в этом случае дерево перестраивать не надо.

Пусть теперь характеристика A до перестраивания была равна -1 и новая вершина добавлена к левой ветви A (аналогично – для случая **1** и добавления к правой ветви). Рассмотрим вершину B – левого потомка A. Возможны следующие варианты.

Если характеристика B после добавления новой вершины в D стала равна -1, то дерево имеет структуру, изображенную на рис. 7.6, а. Перестроив дерево так, как это изображено на рис. 7.6, б, мы добьемся сбалансированности (в скобках указаны характеристики вершин, где это существенно, и соотношения высот после добавления).

Если характеристика вершины B после добавления новой вершины в E стала равна 1, то надо отдельно рассмотреть случаи, когда характеристика вершины E, следующей за B на выделенном пути, стала равна -1, 1 и 0 (в последнем случае вершина E – новая). Вид дерева до и после перестройки для этих случаев показан соответственно на рис. 7.7, 7.8 и 7.9.

7.6 Реализация блочной структуры

С точки зрения структуры программы блоки (и/или процедуры) образуют дерево. Каждой вершине дерева этого представления, соответствующей блоку, можно сопоставить свою таблицу символов (и, возможно, одну общую таблицу идентификаторов). Работу с таблицами блоков можно организовать в магазинном режиме: при входе в блок создавать таблицу символов, при выходе – уничтожать. При этом сами таблицы должны быть связаны в упорядоченный список, чтобы можно было просматривать их в порядке вложенности. Если таблицы организованы с помощью функций расстановки, это означает, что для каждой таблицы должна быть создана своя таблица расстановки.

7.7 Сравнение методов реализации таблиц

Рассмотрим преимущества и недостатки рассмотренных методов реализации таблиц с точки зрения техники использования памяти.

Использование динамической памяти, как правило, довольно дорогая операция, поскольку механизмы поддержания работы с динамической памятью могут быть достаточно сложны. Необходимо поддерживать списки свободной и занятой памяти, выбирать наиболее подходящий кусок памяти при запросе, включать освободившийся кусок в список свободной памяти и, возможно, склеивать куски свободной памяти в списке.

С другой стороны, использование массива требует отведения заранее довольно большой памяти, а это означает, что значительная память вообще не будет использоваться. Кроме того, часто приходится заполнять не все элементы массива (например, в таблице идентификаторов или в тех случаях, когда в массиве фактически хранятся записи переменной длины, например, если в таблице символов записи для различных объектов имеют различный состав полей). Обращение к элементам массива может означать использование операции умножения при вычислении индексов, что может замедлить исполнение.

Наилучшим, по-видимому, является механизм доступа по указателям и использование факта магазинной организации памяти в компиляторе. Для этого процедура выделения памяти выдает необходимый кусок из подряд идущей памяти, а при выходе из процедуры вся память, связанная с этой процедурой, освобождается простой перестановкой указателя свободной памяти в состояние перед началом обработки процедуры. В чистом виде это не всегда, однако, возможно. Например, локальный модуль в Модуле-2 может экспортировать некоторые объекты наружу. При этом схему реализации приходится “подгонять” под механизм распределения памяти. В данном случае, например, необходимо экспортированные объекты вынести в среду охватывающего блока и свернуть блок локального модуля.

Глава 8

Промежуточное представление программы

В процессе трансляции компилятор часто используют промежуточное представление (ПП) исходной программы, предназначенное прежде всего для удобства генерации кода и/или проведения различных оптимизаций. Сама форма ПП зависит от целей его использования.

Наиболее часто используемыми формами ПП является ориентированный граф (в частности, абстрактное синтаксическое дерево, в том числе атрибутированное), трехадресный код (в виде троек или четверок), префиксная и постфиксная запись.

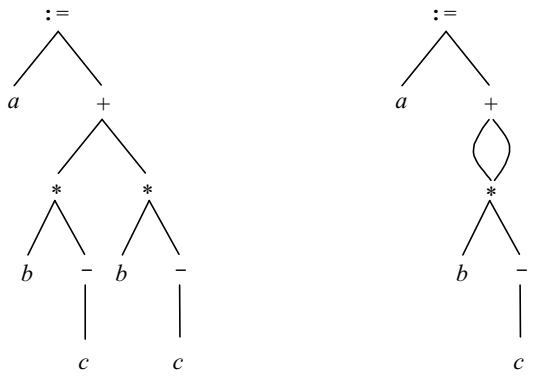
8.1 Представление в виде ориентированного графа

Простейшей формой промежуточного представления является синтаксическое дерево программы. Ту же самую информацию о входной программе, но в более компактной форме дает ориентированный ациклический граф (ОАГ), в котором в одну вершину объединены вершины синтаксического дерева, представляющие общие подвыражения. Синтаксическое дерево и ОАГ для оператора присваивания

$$a := b * -c + b * -c$$

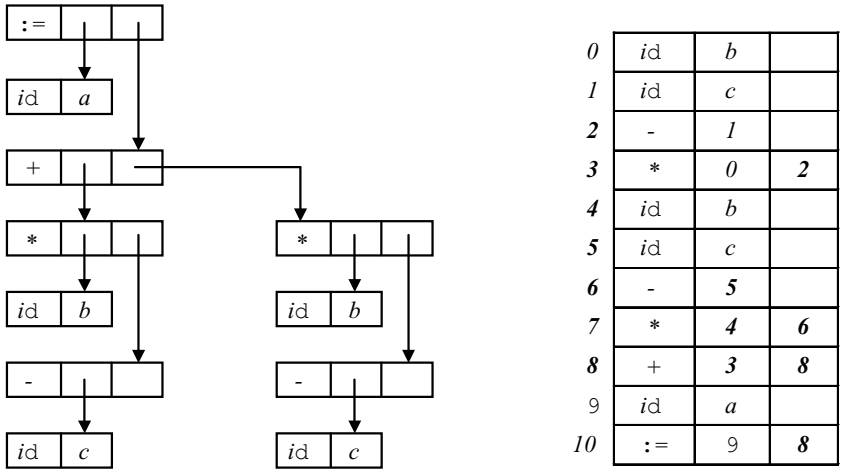
приведены на рис. 8.1.

На рис. 8.2 приведены два представления в памяти синтаксического дерева на рис. 8.1, а. Каждая вершина кодируется записью с полем для операции и полями для указателей на потомков. На рис. 8.2, б, вершины размещены в массиве записей и индекс (или вход) вершины служит указателем на нее.



а *б*

Рис. 8.1:



а *б*

Рис. 8.2:

8.2 Трехадресный код

Трехадресный код – это последовательность операторов вида $x := y\ op\ z$, где x, y и z – имена, константы или сгенерированные компилятором временные объекты. Здесь op – двуместная операция, например операция плавающей или фиксированной арифметики, логическая или побито-

вая. В правую часть может входить только один знак операции.

Составные выражения должны быть разбиты на подвыражения, при этом могут появиться временные имена (переменные). Смысл термина “трехадресный код” в том, что каждый оператор обычно имеет три адреса: два для операндов и один для результата. Трехадресный код – это линеаризованное представление синтаксического дерева или ОАГ, в котором временные имена соответствуют внутренним вершинам дерева или графа. Например, выражение $x + y * z$ может быть протранслировано в последовательность операторов

```
t1 := y * z
t2 := x + t1
```

где t1 и t2 – имена, сгенерированные компилятором.

В виде трехадресного кода представляются не только двуместные операции, входящие в выражения. В таком же виде представляются операторы управления программы и одноместные операции. В этом случае некоторые из компонент трехадресного кода могут не использоваться. Например, условный оператор

```
if A > B then S1 else S2
```

может быть представлен следующим кодом:

```
t := A - B
JGT t, S2
 ...
```

Здесь JGT – двуместная операция условного перехода, не вырабатывающая результата.

Разбиение арифметических выражений и операторов управления делает трехадресный код удобным при генерации машинного кода и оптимизации. Использование имен промежуточных значений, вычисляемых в программе, позволяет легко переупорядочивать трехадресный код.

```
t1 := -c          t1 := -c
t2 := b * t1       t2 := b * t1
t3 := -c          t5 := t2 + t2
t4 := b * t3        a := t5
t5 := t2 + t4
 a := t5
```

а б

Рис. 8.3:

Представления синтаксического дерева и графа рис. 8.1 в виде трехадресного кода дано на рис. 8.3, а, и 8.3, б, соответственно.

Трехадресный код – это абстрактная форма промежуточного кода. В реализации трехадресный код может быть представлен записями с полями для операции и операндов. Рассмотрим три способа реализации трехадресного кода: четверки, тройки и косвенные тройки.

Четверка – это запись с четырьмя полями, которые будем называть op, $arg1$, $arg2$ и $result$. Поле op содержит код операции. В операторах с унарными операциями типа $x := -y$ или $x := y$ поле $arg2$ не используется. В некоторых операциях (типа "передать параметр") могут не использоваться ни $arg2$, ни $result$. Условные и безусловные переходы помещают в $result$ метку перехода. На рис. 8.4, а, приведены четверки для оператора присваивания $a := b*-c+b*-c$. Они получены из трехадресного кода на рис. 8.3, а.

	op	$arg1$	$arg2$	$result$
(0)	-	c		t1
(1)	*	b	t1	t2
(2)	-	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

а) четверки

	op	$arg1$	$arg2$
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

б) тройки

Рис. 8.4:

Обычно содержимое полей $arg1$, $arg2$ и $result$ – это указатели на входы таблицы символов для имен, представляемых этими полями. Временные имена вносятся в таблицу символов по мере их генерации.

Чтобы избежать внесения новых имен в таблицу символов, на временное значение можно ссылаться, используя позицию вычисляющего его оператора. В этом случае трехадресные операторы могут быть представлены записями только с тремя полями: op, $arg1$ и $arg2$, как это показано на рис. 8.3, б. Поля $arg1$ и $arg2$ – это либо указатели на таблицу символов (для имен, определенных программистом, или констант), либо указатели на тройки (для временных значений). Такой способ представления трехадресного кода называют *тройками*. Тройки соответствуют представлению синтаксического дерева или ОАГ с помощью массива вершин.

Числа в скобках – это указатели на тройки, а имена – это указатели на таблицу символов. На практике информация, необходимая для интерпретации различного типа входов в поля $arg1$ и $arg2$, кодируется в поле op или дополнительных полях. Тройки рис. 8.4, б, соответствуют четверкам рис. 8.4, а.

Для представления тройками трехместной операции типа $x[i] := y$ требуется два входа, как это показано на рис. 8.5, а, представление $x := y[i]$ двумя операциями показано на рис. 8.5, б.

	op	$arg1$	$arg2$
(0)	[ ]=	x	i
(1)	:=	(0)	y

а) $x[i] := y$

	op	$arg1$	$arg2$
(0)	=[ ]	y	i
(1)	:=	x	(0)

б) $x := y[i]$

Рис. 8.5:

Трехадресный код может быть представлен не списком троек, а списком указателей на них. Такая реализация обычно называется *косвенными тройками*. Например, тройки рис. 8.4, б, могут быть реализованы так, как это изображено на рис. 8.6.

	оператор
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	$arg1$	$arg2$
(14)	-	c	
(15)	*	b	(14)
(16)	-	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Рис. 8.6:

При генерации объектного кода каждой переменной, как временной, так и определенной в исходной программе, назначается память периода исполнения, адрес которой обычно хранится в таблице генератора кода. При использовании четверок этот адрес легко получить через эту таблицу.

Более существенно преимущество четверок проявляется в оптимизирующих компиляторах, когда может возникнуть необходимость перемещать операторы. Если перемещается оператор, вычисляющий x, не требуется изменений в операторе, использующем x. В записи же тройками перемещение оператора, определяющего временное значение, требует изменения всех ссылок на этот оператор в массивах $arg1$ и $arg2$. Из-за этого тройки трудно использовать в оптимизирующих компиляторах.

В случае применения косвенных троек оператор может быть перемещен переупорядочиванием списка операторов. При этом не надо менять указатели на *op*, $arg1$ и $arg2$. Этим косвенные тройки похожи на четверки. Кроме того, эти два способа требуют примерно одинаковой памяти. Как и в случае простых троек, при использовании косвенных троек выделение памяти для временных значений может быть отложено на этап генерации кода. По сравнению с четверками при использование косвенных троек можно сэкономить память, если одно и то же временное значение используется более одного раза. Например, на рис. 8.6 можно объ-

единить строки (14) и (16), после чего можно объединить строки (15) и (17).

8.3 Линеаризованные представления

В качестве промежуточных представлений весьма распространены линеаризованные представления деревьев. Линеаризованное представление позволяет относительно легко хранить промежуточное представление во внешней памяти и обрабатывать его. Наиболее распространенной формой линеаризованного представления является *польская запись* – префиксная (прямая) или постфиксная (обратная).

Постфиксная запись – это список вершин дерева, в котором каждая вершина следует (при обходе снизу-вверх слева-направо) непосредственно за своими потомками. Дерево на рис. 8.1, а, в постфиксной записи может быть представлено следующим образом:

```
a b c - * b c - * + :=
```

В постфиксной записи вершины синтаксического дерева явно не присутствуют. Они могут быть восстановлены из порядка, в котором следуют вершины и из числа операндов соответствующих операций. Восстановление вершин аналогично вычислению выражения в постфиксной записи с использованием стека.

В префиксной записи сначала указывается операция, а затем ее операнды. Например, для приведенного выше выражения имеем

```
:= a + * b - c * b - c
```

Рассмотрим детальнее одну из реализаций префиксного представления – Лидер [9]. Лидер – это аббревиатура от “ЛИнеаризованное ДЕРево”. Это машинно-независимая префиксная запись. В Лидере сохраняются все объявления и каждому из них присваивается свой уникальный номер, который используется для ссылки на объявление. Рассмотрим пример.

```
module M;
var X,Y,Z: integer;
procedure DIF(A,B:integer):integer;
  var R:integer;
  begin R:=A-B;
      return(R);
  end DIF;
begin Z:=DIF(X,Y);
end M.
```

Этот фрагмент имеет следующий образ в Лидере.

```
program 'M'
var int
var int
var int
procbody proc int int end int
  var int
  begin assign var 1 7 end
            int int mi par 1 5 end par 1 6 end
      result 0 int var 1 7 end
      return
  end
begin assign var 0 3 end int
    icall 0 4 int var 0 1 end
    int var 0 2 end end
end
```

Рассмотрим его более детально:

```
program 'M'        Имя модуля нужно для редактора связей.
var int            Это образ переменных X, Y, Z;
var int            переменным X, Y, Z присваиваются номера
var int            1, 2, 3 на уровне 0.
procbody proc      Объявление процедуры с двумя
int int end        целыми параметрами, возвращающей целое.
int                Процедура получает номер 4 на уровне 0 и
                   параметры имеют номера 5, 6 на уровне 1.
var int            Переменная R имеет номер 7 на уровне 1.
begin              Начало тела процедуры.
assign             Оператор присваивания.
var 1 7 end        Левая часть присваивания (R).
int                Тип присваиваемого значения.
int mi             Целое вычитание.
par 1 5 end        Уменьшаемое (A).
par 1 6 end        Вычитаемое (B).
result 0           Результат процедуры уровня 0.
int                Результат имеет целый тип.
var 1 7 end        Результат – переменная R.
return             Оператор возврата.
end                Конец тела процедуры.
begin              Начало тела модуля.
assign             Оператор присваивания.
var 0 3 end        Левая часть – переменная Z.
int                Тип присваиваемого значения.
icall 0 4          Вызов локальной процедуры DIF.
int var 0 1 end    Фактические параметры X
int var 0 2 end    и Y.
end                Конец вызова.
end                Конец тела модуля.
```

8.4 Виртуальная машина Java

Программы на языке Java транслируются в специальное промежуточное представление, которое затем интерпретируется так называемой “виртуальной машиной Java”. Виртуальная машина Java представляет собой стековую машину: она не имеет памяти прямого доступа, все операции выполняются над операндами, расположенными на верхушке стека. Чтобы, например, выполнить операцию с участием константы или переменной, их предварительно необходимо загрузить на верхушку стека. Код операции – всегда один байт. Если операция имеет операнды, они располагаются в следующих байтах.

К элементарным типам данных, с которыми работает машина, относятся short, integer, long, float, double (все знаковые).

8.4.1 Организация памяти

Машина имеет следующие регистры:

pc – счетчик команд;
optop – указатель вершины стека операций;
frame – указатель на стек-фрейм исполняемого метода;
vars – указатель на 0-ю переменную исполняемого метода.

Все регистры 32-разрядные. Стек-фрейм имеет три компоненты: локальные переменные, среду исполнения, стек операндов. Локальные переменные отсчитываются от адреса в регистре vars. Среда исполнения служит для поддержания самого стека. Она включает указатель на предыдущий фрейм, указатель на собственные локальные переменные, на базу стека операций и на верхушку стека. Кроме того, здесь же хранится некоторая дополнительная информация, например, для отладчика.

Куча сборки мусора содержит экземпляры объектов, которые создаются и уничтожаются автоматически. Область методов содержит коды, таблицы символов и т.д.

С каждым классом связана область констант. Она содержит имена полей, методов и другую подобную информацию, которая используется методами.

8.4.2 Набор команд виртуальной машины

Виртуальная Java-машина имеет следующие команды:

помещение констант на стек,
помещение локальных переменных на стек,
запоминание значений из стека в локальных переменных,
обработка массивов,
управление стеком,
арифметические команды,
логические команды,
преобразования типов,
передача управления,
возврат из функции,
табличный переход,
обработка полей объектов,
вызов метода,
обработка исключительных ситуаций,
прочие операции над объектами,
мониторы,
отладка.

Рассмотрим некоторые команды подробнее.

Помещение локальных переменных на стек

Команда iload – загрузить целое из локальной переменной. Операндом является смещение переменной в области локальных переменных. Указываемое значение копируется на верхушку стека операций. Имеются аналогичные команды для помещения плавающих, двойных целых, двойных плавающих и т.д.

Команда istore – сохранить целое в локальной переменной. Операндом операции является смещение переменной в области локальных переменных. Значение с верхушки стека операций копируется в указываемую область локальных переменных. Имеются аналогичные команды для помещения плавающих, двойных целых, двойных плавающих и т.д.

Вызов метода

Команда invokevirtual.

При трансляции объектно-ориентированных языков программирования из-за возможности перекрытия виртуальных методов, вообще говоря, нельзя статически протранслировать вызов метода объекта. Это связано с тем, что если метод перекрыт в производном классе, и вызывается метод объекта-переменной, то статически неизвестно, объект какого класса (базового или производного) хранится в переменной. Поэтому с каждым объектом связывается таблица всех его виртуальных методов: для каждого метода там помещается указатель на его реализацию в соответствии с принадлежностью самого объекта классу в иерархии классов.

В языке Java различные классы могут реализовывать один и тот же интерфейс. Если объявлена переменная или параметр типа интерфейс, то динамически нельзя определить объект какого класса присвоен переменной:

```
interface I;
class C1 implements I;
class C2 implements I;
I O;
C1 O1;
C2 O2;
...
O=O1;
...
O=O2;
...
```

В этой точке программы, вообще говоря, нельзя сказать, какого типа значение хранится в переменной O. Кроме того, при работе программы на языке Java имеется возможность использования методов из других

пакетов. Для реализации этого механизма в Java-машине используется динамическое связывание.

Предполагается, что стек операндов содержит handle объекта или массива и некоторое количество аргументов. Операнд операции используется для конструирования индекса в области констант текущего класса. Элемент по этому индексу в области констант содержит полную сигнатуру метода. Сигнатура метода описывает типы параметров и возвращаемого значения. Из handle объекта извлекается указатель на таблицу методов объекта. Просматривается сигнатура метода в таблице методов. Результатом этого просмотра является индекс в таблицу методов именованного класса, для которого найден указатель на блок метода. Блок метода указывает тип метода (native, synchronized и т.д.) и число аргументов, ожидаемых на стеке операндов.

Если метод помечен как synchronized, запускается монитор, связанный с handle. Базис массива локальных переменных для нового стек-фрейма устанавливается так, что он указывает на handle на стеке. Определяется общее число локальных переменных, используемых методом, и после того, как отведено необходимое место для локальных переменных, окружение исполнения нового фрейма помещается на стек. База стека операндов для этого вызова метода устанавливается на первое слово после окружения исполнения. Затем исполнение продолжается с первой инструкции вызванного метода.

Обработка исключительных ситуаций

Команда athrow – возбудить исключительную ситуацию.

С каждым методом связан список операторов catch. Каждый оператор catch описывает диапазон инструкций, для которых он активен, тип исключения, который он обрабатывает. Кроме того, с оператором связан набор инструкций, которые его реализуют. При возникновении исключительной ситуации просматривается список операторов catch, чтобы установить соответствие. Исключительная ситуация соответствует оператору catch, если инструкция, вызвавшая исключительную ситуацию, находится в соответствующем диапазоне и исключительная ситуация принадлежит подтипу типа ситуации, которые обрабатывает оператор catch. Если соответствующий оператор catch найден, управление передается обработчику. Если нет, текущий стек-фрейм удаляется, и исключительная ситуация возбуждается вновь.

Порядок операторов catch в списке важен. Интерпретатор передает управление первому подходящему оператору catch.

8.5 Организация информации в генераторе кода

Синтаксическое дерево в чистом виде несет только информацию о структуре программы. На самом деле в процессе генерации кода требуется также информация о переменных (например, их адреса), процедурах (также адреса, уровни), метках и т.д. Для представления этой информации возможны различные решения. Наиболее распространены два:

– информация хранится в таблицах генератора кода;

– информация хранится в соответствующих вершинах дерева.

Рассмотрим, например, структуру таблиц, которые могут быть использованы в сочетании с Лидер-представлением. Поскольку Лидер-представление не содержит информации об адресах переменных, значит, эту информацию нужно формировать в процессе обработки объявлений и хранить в таблицах. Это касается и описаний массивов, записей и т.д. Кроме того, в таблицах также должна содержаться информация о процедурах (адреса, уровни, модули, в которых процедуры описаны, и т.д.).

При входе в процедуру в таблице уровней процедур заводится новый вход – указатель на таблицу описаний. При выходе указатель восстанавливается на старое значение. Если промежуточное представление – дерево, то информация может храниться в вершинах самого дерева.

8.6 Уровень промежуточного представления

Как видно из приведенных примеров, промежуточное представление программы может в различной степени быть близким либо к исходной программе, либо к машине. Например, промежуточное представление может содержать адреса переменных, и тогда оно уже не может быть перенесено на другую машину. С другой стороны, промежуточное представление может содержать раздел описаний программы, и тогда информацию об адресах можно извлечь из обработки описаний. В то же время ясно, что первое более эффективно, чем второе. Операторы управления в промежуточном представлении могут быть представлены в исходном виде (в виде операторов языка if, for, while и т.д.), а могут содержаться в виде переходов. В первом случае некоторая информация может быть извлечена из самой структуры (например, для оператора for – информация о переменной цикла, которую, может быть, разумно хранить на регистре, для оператора case – информация о таблице меток и т.д.). Во втором случае представление проще и унифицированней.

Некоторые формы промежуточного представления удобны для различного рода оптимизаций, некоторые – нет (например, косвенные тройки, в отличие от префиксной записи, позволяют эффективное перемещение кода).

Глава 9

Генерация кода

Задача генератора кода – построение для программы на входном языке эквивалентной машинной программы. Обычно в качестве входа для генератора кода служит некоторое промежуточное представление программы.

Генерация кода включает ряд специфических, относительно независимых подзадач: распределение памяти (в частности, распределение регистров), выбор команд, генерацию объектного (или загрузочного) модуля. Конечно, независимость этих подзадач относительна: например, при выборе команд нельзя не учитывать схему распределения памяти, и, наоборот, схема распределения памяти (регистров, в частности) ведет к генерации той или иной последовательности команд. Однако удобно и практично эти задачи все же разделять, обращая при этом внимание на их взаимодействие.

В какой-то мере схема генератора кода зависит от формы промежуточного представления. Ясно, что генерация кода из дерева отличается от генерации кода из троек, а генерация кода из префиксной записи отличается от генерации кода из ориентированного графа. В то же время все генераторы кода имеют много общего, и основные применяемые алгоритмы отличаются, как правило, только в деталях, связанных с используемым промежуточным представлением.

В дальнейшем в качестве промежуточного представления мы будем использовать префиксную нотацию. А именно, алгоритмы генерации кода будем излагать в виде атрибутных схем со входным языком Лидер.

9.1 Модель машины

При изложении алгоритмов генерации кода мы будем следовать некоторой модели машины, в основу которой положена система команд микропроцессора Motorola MC68020. В микропроцессоре имеется регистр – счетчик команд PC, 8 регистров данных и 8 адресных регистров.

В системе команд используются следующие способы адресации:

ABS – абсолютная: исполнительным адресом является значение адресного выражения.

IMM – непосредственный операнд: операндом команды является константа, заданная в адресном выражении.

D – прямая адресация через регистр данных, записывается как Xn, операнд находится в регистре Xn.

A – прямая адресация через адресный регистр, записывается как An, операнд находится в регистре An.

INDIRECT – записывается как (An), адрес операнда находится в адресном регистре An.

POST – пост-инкрементная адресация, записывается как (An)+, исполнительный адрес есть значение адресного регистра An и после исполнения команды значение этого регистра увеличивается на длину операнда.

PRE – пре-инкрементная адресация, записывается как -(An): перед исполнением операции содержимое адресного регистра An уменьшается на длину операнда, исполнительный адрес равен новому содержимому адресного регистра.

INDISP – косвенная адресация со смещением, записывается как (bd,An), исполнительный адрес вычисляется как (An)+d – содержимое An плюс d.

INDEX – косвенная адресация с индексом, записывается как (bd,An,Xn*sc), исполнительный адрес вычисляется как (An)+bd+(Xn)*sc – содержимое адресного регистра + адресное смещение + содержимое индексного регистра, умноженное на sc.

INDIRPC – косвенная через PC (счетчик команд), записывается как (bd,PC), исполнительный адрес определяется выражением (PC)+bd.

INDEXPC – косвенная через PC со смещением, записывается как (bd,PC,Xn*sc), исполнительный адрес определяется выражением (PC)+bd+(Xn)*sc.

INDPRE – пре-косвенная через память, записывается как ([bd,An,sc*Xn],od) (схема вычисления адресов для этого и трех последующих способов адресации приведена ниже).

INDPOST – пост-косвенная через память: ([bd,An],sc*Xn,od).

INDPREPC – пре-косвенная через PC: ([bd,PC,sc*Xn],od).

INDPOSTPC – пост-косвенная через PC: ([bd,PC],Xn,od).

Здесь bd – это 16- или 32-битная константа, называемая *смещением*, od – 16- или 32-битная литеральная константа, называемая *внешним смещением*. Эти способы адресации могут использоваться в упрощенных формах без смещений bd и/или od и без регистров An или Xn. Следующие примеры иллюстрируют косвенную постиндексную адресацию:

```
MOVE D0, ([A0])
MOVE D0, ([4,A0])
MOVE D0, ([A0],6)
MOVE D0, ([A0],D3)
MOVE D0, ([A0],D4,12)
```

```
MOVE D0, ([$12345678,A0],D4,$FF000000)
```

Индексный регистр Xn может масштабироваться (умножаться) на 2,4,8, что записывается как sc*Xn. Например, в исполнительном адресе ([24,A0,4*D0]) содержимое квадратных скобок вычисляется как [A0] + 4 * [D0] + 24.

Эти способы адресации работают следующим образом. Каждый исполнительный адрес содержит пару квадратных скобок [...] внутри пары круглых скобок, т.е. ([...],...). Сначала вычисляется содержимое квадратных скобок, в результате чего получается 32-битный указатель. Например, если используется постиндексная форма [20,A2], то исполнительный адрес – это 20 + [A2]. Аналогично, для преиндексной формы [12,A4,D5] исполнительный адрес – это 12 + [A4] + [D5].

Указатель, сформированный содержимым квадратных скобок, используется для доступа в память, чтобы получить новый указатель (отсюда термин косвенная адресация через память). К этому новому указателю добавляется содержимое внешних круглых скобок и таким образом формируется исполнительный адрес операнда.

В дальнейшем изложении будут использованы следующие команды (в частности, рассматриваются только арифметические команды с целыми операндами, но не с плавающими):

MOVEA ИА, A – загрузить содержимое по исполнительному адресу ИА на адресный регистр A.

MOVE ИА1, ИА2 – содержимое по исполнительному адресу ИА1 переписать по исполнительному адресу ИА2.

MOVEM список_регистров, ИА – сохранить указанные регистры в памяти, начиная с адреса ИА (регистры указываются маской в самой команде).

MOVEM ИА, список_регистров – восстановить указанные регистры из памяти, начиная с адреса ИА (регистры указываются маской в самой команде).

LEA ИА, A – загрузить исполнительный адрес ИА на адресный регистр A.

MUL ИА, D – умножить содержимое по исполнительному адресу ИА на содержимое регистра данных D и результат разместить в D (на самом деле в системе команд имеются две различные команды MULS и MULU для чисел со знаком и чисел без знака соответственно; для упрощения мы не будем принимать во внимание это различие).

DIV ИА, D – разделить содержимое регистра данных D на содержимое по исполнительному адресу ИА и результат разместить в D.

ADD ИА, D – сложить содержимое по исполнительному адресу ИА с содержимым регистра данных D и результат разместить в D.

SUB ИА, D – вычесть содержимое по исполнительному адресу ИА из содержимого регистра данных D и результат разместить в D.

Команды CMP и TST формируют разряды регистра состояний. Всего имеется 4 разряда: Z – признак нулевого результата, N – признак отри-

цательного результата, V – признак переполнения, C – признак переноса.

CMP ИА, D – из содержимого регистра данных D вычитается содержимое по исполнительному адресу ИА, при этом формируется все разряды регистра состояний, но содержимое регистра D не меняется.

TST ИА – выработать разряд Z регистра состояний по значению, находящемуся по исполнительному адресу ИА.

BNE ИА – условный переход по признаку $Z = 1$ (не равно) по исполнительному адресу ИА.

BEQ ИА – условный переход по признаку $Z = 0$ (равно) по исполнительному адресу ИА.

BLE ИА – условный переход по признаку N *or* Z (меньше или равно) по исполнительному адресу ИА.

BGT ИА – условный переход по признаку $not\ N$ (больше) по исполнительному адресу ИА.

BLT ИА – условный переход по признаку N (меньше) по исполнительному адресу ИА.

BRA ИА – безусловный переход по адресу ИА.

JMP ИА – безусловный переход по исполнительному адресу.

RTD размер_локальных – возврат из подпрограммы с указанием размера локальных.

LINK A, размер_локальных – в стеке сохраняется значение регистра A, в регистр A заносится указатель на это место в стеке и указатель стека продвигается на размер локальных.

UNLK A – стек сокращается на размер локальных и регистр A восстанавливается из стека.

9.2 Динамическая организация памяти

Динамическая организация памяти - это организация памяти периода исполнения программы. Оперативная память программы обычно состоит из нескольких основных разделов: стек (магазин), куча, область статических данных (инициализированных и неинициализированных). Наиболее сложной является работа со стеком. Вообще говоря, стек периода исполнения необходим для программ не на всех языках программирования. Например, в ранних версиях Фортрана нет рекурсии, так что программа может исполняться без стека. С другой стороны, исполнение программы с рекурсией может быть реализовано и без стека (того же эффекта можно достичь, например, и с помощью списковых структур). Однако, для эффективной реализации пользуются стеком, который, как правило, поддерживается на уровне машинных команд.

Рассмотрим схему организации магазина периода выполнения для простейшего случая (как, например, в языке Паскаль), когда все переменные в магазине (фактические параметры и локальные переменные)

имеют известные при трансляции смещения. Магазин служит для хранения локальных переменных (и параметров) и обращения к ним в языках, допускающих рекурсивные вызовы процедур. Еще одной задачей, которую необходимо решать при трансляции языков с блочной структурой – обеспечение реализации механизмов статической вложенности. Пусть имеется следующий фрагмент программы на Паскале:

```
procedure P1;
 var V1;
 procedure P2;
  var V2;
 begin
  ...
  P2;
  V1:=...
  V2:=...
  ...
 end;
begin
 ...
 P2;
 ...
end;
```

В процессе выполнения этой программы, находясь в процедуре P2, мы должны иметь доступ к последнему экземпляру значений переменных процедуры P2 и к экземпляру значений переменных процедуры P1, из которой была вызвана P2. Кроме того, необходимо обеспечить восстановление состояния программы при завершении выполнения процедуры.

Мы рассмотрим две возможные схемы динамической организации памяти: схему со статической цепочкой и с дисплеем в памяти. В первом случае все статические контексты связаны в список, который называется *статической цепочкой*; в каждой записи для процедуры в магазине хранится указатель на запись статически охватывающей процедуры (помимо, конечно, указателя *динамической цепочки* – указателя на "базу" динамически предыдущей процедуры). Во втором случае для хранения ссылок на статические контексты используется массив, называемый *дисплеем*. Использование той или иной схемы определяется, помимо прочих условий, прежде всего числом адресных регистров.

9.2.1 Организация магазина со статической цепочкой

Итак, в случае статической цепочки магазин организован, как это изображено на рис. 9.1.

Таким образом, на запись текущей процедуры в магазине указывает регистр BP (Base Pointer), с которого начинается динамическая цепочка.

Рис. 9.1:

На статическую цепочку указывает регистр LP (Link Pointer). В качестве регистров BP и LP в различных системах команд могут использоваться универсальные, адресные или специальные регистры. Локальные переменные отсчитываются от регистра BP вверх, фактические параметры – вниз с учетом памяти, занятой точкой возврата и самим сохраненным регистром BP.

Вызов подпрограмм различного статического уровня производится несколько по-разному. При вызове подпрограммы того же статического уровня, что и вызывающая подпрограмма (например, рекурсивный вызов той же самой подпрограммы), выполняются следующие команды:

```
Занесение фактических параметров в магазин
JSR A
```

Команда JSR A продвигает указатель SP, заносит PC на верхушку магазина и осуществляет переход по адресу A. После выполнения этих команд состояние магазина становится таким, как это изображено на рис. 9.2.

Занесение BP, отведение локальных, сохранение регистров делает вызываемая подпрограмма (см. ниже).

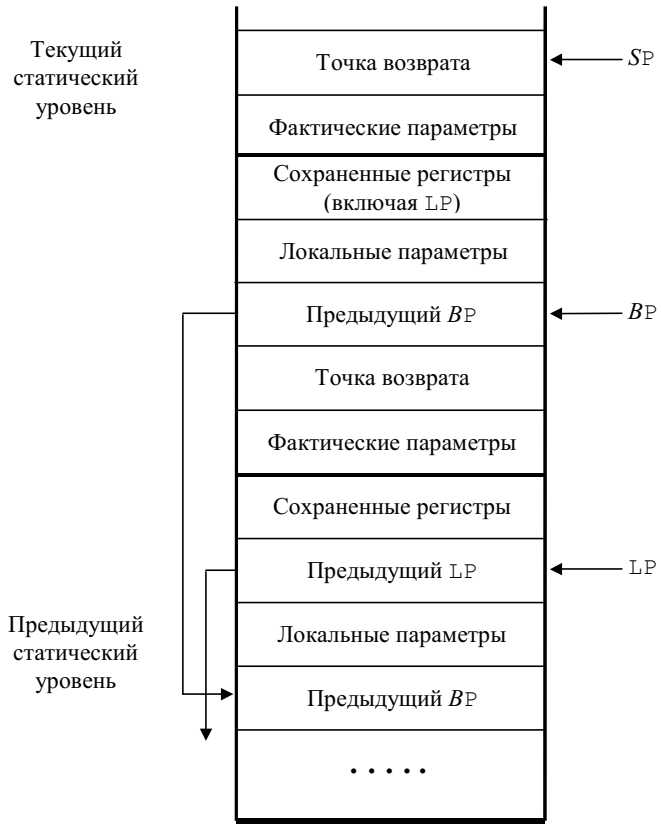


Рис. 9.2:

При вызове локальной подпрограммы необходимо установить указатель статического уровня на текущую подпрограмму, а при выходе – восстановить его на старое значение (охватывающей текущую). Для этого исполняются следующие команды:

```
Занесение фактических параметров в магазин
MOVE BP, LP
SUB Delta, LP
JSR A
```

Здесь Delta – размер локальных вызывающей подпрограммы плюс двойная длина слова. Магазин после этого принимает состояние, изображенное на рис. 9.3. Предполагается, что регистр LP уже сохранен

среди сохраняемых регистров, причем самым первым (сразу после локальных переменных).

После выхода из подпрограммы в вызывающей подпрограмме выполняется команда

```
MOVE (LP), LP
```

которая восстанавливает старое значение статической цепочки. Если выход осуществлялся из подпрограммы 1-го уровня, эту команду выполнять не надо, поскольку для 1-го уровня нет статической цепочки.

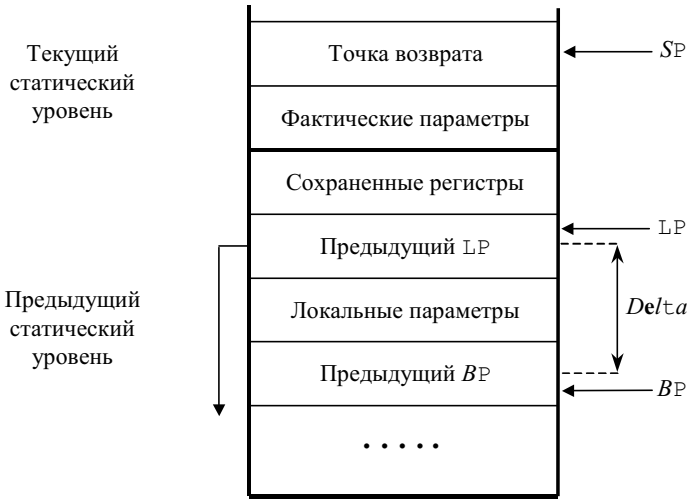


Рис. 9.3:

При вызове подпрограммы меньшего, чем вызывающая, уровня выполняются следующие команды:

```
Занесение фактических параметров в магазин
MOVE (LP), LP /* столько раз, какова разность
        уровней вызывающей и вызываемой ПП */
JSR A
```

Тем самым устанавливается статический уровень вызываемой подпрограммы. После выхода из подпрограммы выполняется команда

```
MOVE -Delta(BP), LP
```

восстанавливающая статический уровень вызывающей подпрограммы.

Тело подпрограммы начинается со следующих команд:

```
LINK BP , -размер_локальных
MOVEM -(SP)
```

Команда LINK BP, размер_локальных эквивалентна трем командам:

```
MOVE BP, -(SP)
MOVE SP, BP
ADD -размер_локальных, SP
```

Команда MOVEM сохраняет в магазине регистры.

В результате выполнения этих команд магазин приобретает вид, изображенный на рис. 9.1.

Выход из подпрограммы осуществляется следующей последовательностью команд:

```
MOVEM (SP)+
UNLK BP
RTD размер_фактических
```

Команда MOVEM восстанавливает регистры из магазина. Команда UNLK BP эквивалентна такой последовательности команд:

```
MOVE BP,SP
MOVE (SP),BP
ADD #4, SP  /* 4 - размер слова */
```

Команда RTD размер_фактических, в свою очередь, эквивалентна последовательности

```
ADD размер_фактических+4, SP
JMP -размер_фактических-4(SP)
```

После ее выполнения магазин восстанавливается до состояния, которое было до вызова.

В зависимости от наличия локальных переменных, фактических параметров и необходимости сохранения регистров каждая из этих команд может отсутствовать.

9.2.2 Организация магазина с дисплеем

Рассмотрим теперь организацию магазина с дисплеем. Дисплей – это массив (DISPLAY) , i-й элемент которого представляет собой указатель на область активации последней вызванной подпрограммы i-го статического уровня. Доступ к переменным самой внутренней подпрограммы осуществляется через регистр BP. Дисплей может быть реализован либо через регистры (если их достаточно), либо через массив в памяти.

При вызове процедуры следующего (по отношению к вызывающей) уровня в дисплее отводится очередной элемент. Если вызывающая процедура имеет статический уровень i, то при вызове процедуры уровня $j \leqslant i$ элементы дисплея $j, \ldots, i$ должны быть скопированы (обычно в стек вызывающей процедуры), текущим уровнем становится j и в DISPLAY[j] заносится указатель на область активации вызываемой процедуры. По окончании работы вызываемой процедуры содержимое дисплея восстанавливается из стека.

Иногда используется комбинированная схема – дисплей в магазине. Дисплей хранится в области активации каждой процедуры. Формирование дисплея для процедуры осуществляется в соответствии с правилами, описанными выше.

Отдельного рассмотрения требует вопрос о технике передачи фактических параметров. Конечно, в случае простых параметров (например, чисел) проблем не возникает. Однако передача массивов по значению – операция довольно дорогая, поэтому с точки зрения экономии памяти целесообразнее сначала в подпрограмму передать адрес массива, а затем уже из подпрограммы по адресу передать в магазин сам массив. В связи с передачей параметров следует упомянуть еще одно обстоятельство.

Рассмотренная схема организации магазина допустима только для языков со статически известными размерами фактических параметров. Однако, например, в языке Модула-2 по значению может быть передан гибкий массив, и в этом случае нельзя статически распределить память для параметров. Обычно в таких случаях заводят так называемый “паспорт” массива, в котором хранится вся необходимая информация, а сам массив размещается в магазине в рабочей области выше сохраненных регистров.

9.3 Назначение адресов

Назначение адресов переменным, параметрам и полям записей происходит при обработке соответствующих объявлений. В однопроходном трансляторе это может производиться вместе с построением основной таблицы символов и соответствующие адреса (или смещения) могут храниться в этой же таблице. В промежуточном представлении Лидер объявления сохранены, что делает это промежуточное представление машинно-независимым. Напомним, что в Лидер-представлении каждому описанию соответствует некоторый номер. В процессе работы генератора кодов поддерживается таблица Table, в которой по этому номеру (входу) содержится следующая информация:

– для типа: его размер;

– для переменной: смещение в области процедуры (или глобальной области);

– для поля записи: смещение внутри записи;

– для процедуры: размер локальных параметров;

– для массива: размер массива, размер элемента, значение левой и правой границы.

Функция IncTab вырабатывает указатель (вход) на новый элемент таблицы, проверяя при этом наличие свободной памяти.

Для вычисления адресов определим для каждого объявления два синтезируемых атрибута: DISP будет обозначать смещение внутри области процедуры (или единицы компиляции), а SIZE – размер. Тогда семантика правила для списка объявлений принимает вид

```
RULE
DeclPart ::= ( Decl )
SEMANTICS
  Disp<1>=0;
1A: Disp<1>=Disp<1>+Size<1>;
  Size<0>=Disp<1>.
```

Все объявления, кроме объявлений переменных, имеют нулевой размер. Размер объявления переменной определяется следующим правилом:

```
RULE
Decl ::= 'VAR' TypeDes
SEMANTICS
Tablentry Entry;
0: Entry=IncTab;
  Size<0>=((Table[VAL<2>]+1) / 2)*2;
  // Выравнивание на границу слова
  Table[Entry]=Disp<0>+Size<0>.
```

В качестве примера трансляции определения типа рассмотрим обработку описания записи:

```
RULE
TypeDes ::= 'REC' ( TypeDes ) 'END'
SEMANTICS
int Disp;
Tablentry Temp;
0: Entry<0>=IncTab;
  Disp=0;
2A: {Temp=IncTab;
   Table[Temp]=Disp;
   Disp=Disp+Table[Entry<2>]+1) / 2)*2;
   // Выравнивание на границу слова
   }
Table[Entry<0>]=Disp.
```

9.4 Трансляция переменных

Переменные отражают все многообразие механизмов доступа в языке. Переменная имеет синтезированный атрибут ADDRESS – это запись, описывающая адрес в команде MC68020. Этот атрибут сопоставляется всем нетерминалам, представляющим значения. В системе команд MC68020 много способов адресации, и они отражены в структуре значения атрибута ADDRESS, имеющего следующий тип:

```
enum Register
{D0,D1,D2,D3,D4,D5,D6,D7,A0,A1,A2,A3,A4,A5,A6,SP,N0};

enum AddrMode
{D,A,Post,Pre,Indirect,IndPre,IndPost,IndirPC,
IndPrePC,IndPostPC,InDisp,Index,IndexPC,Abs,Imm};

struct AddrType{
 Register AddrReg,IndexReg;
 int IndexDisp,AddrDisp;
 short Scale;
};
```

Значение регистра N0 означает, что соответствующий регистр в адресации не используется.

Доступ к переменным осуществляется в зависимости от их уровня: глобальные переменные адресуются с помощью абсолютной адресации; переменные в процедуре текущего уровня адресуются через регистр базы A6.

Если стек организован с помощью статической цепочки, то переменные предыдущего статического уровня адресуются через регистр статической цепочки A5; переменные остальных уровней адресуются “пробеганием” по статической цепочке с использованием вспомогательного регистра. Адрес переменной формируется при обработке структуры переменной слева направо и передается сначала сверху вниз как наследуемый атрибут нетерминала VarTail, а затем передается снизу-вверх как глобальный атрибут нетерминала Variable. Таким образом, правило для обращения к переменной имеет вид (первое вхождение Number в правую часть – это уровень переменной, второе – ее Лидер-номер):

```
RULE
Variable ::= VarMode Number Number VarTail
SEMANTICS
int Temp;
struct AddrType AddrTmp1, AddrTmp2;
3: if (Val<2>==0) // Глобальная переменная
    {Address<4>.AddrMode=Abs;
     Address<4>.AddrDisp=0;
```

```
    }
  else // Локальная переменная
    {Address<4>.AddrMode=Index;
     if (Val<2>==Level<Block>) // Переменная текущего уровня
       Address<4>.AddrReg=A6;
     else if (Val<2>==Level<Block>-1)
           // Переменная предыдущего уровня
           Address<4>.AddrReg=A5;
         else
           {Address<4>.Addreg=GetFree(RegSet<Block>);
            AddrTmp1.AddrMode=Indirect;
            AddrTmp1.AddrReg=A5;
            Emit2(MOVEA,AddrTmp1,Address<4>.AddrReg);
            AddrTmp1.AddrReg=Address<4>.AddrReg;
            AddrTmp2.AddrMode=A;
            AddrTmp2.AddrReg=Address<4>.AddrReg;
            for (Temp=Level<Block>-Val<2>;Temp>=2;Temp--)
               Emit2(MOVEA,AddrTmp1,AddrTmp2);
           }
     if (Val<2>==Level<Block>)
       Address<4>.AddrDisp=Table[Val<3>];
     else
       Address<4>.AddrDisp=Table[Val<3>]+Table[LevelTab[Val<2>]];
    }.
```

Функция GetFree выбирает очередной свободный регистр (либо регистр данных, либо адресный регистр) и отмечает его как использованный в атрибуте RegSet нетерминала Block. Процедура Emit2 генерирует двухадресную команду. Первый параметр этой процедуры – код команды, второй и третий параметры имеют тип AddrType и служат операндами команды. Смещение переменной текущего уровня отсчитывается от базы (A6), а других уровней – от указателя статической цепочки, поэтому оно определяется как алгебраическая сумма размера локальных параметров и величины смещения переменной. Таблица LevelTab – это таблица уровней процедур, содержащая указатели на последовательно вложенные процедуры.

Если стек организован с помощью дисплея, то трансляция для доступа к переменным может быть осуществлена следующим образом:

```
RULE
Variable ::= VarMode Number Number VarTail
SEMANTICS
int Temp;
3: if (Val<2>==0) // Глобальная переменная
    {Address<4>.AddrMode=Abs;
     Address<4>.AddrDisp=0;
```

```
   }
 else // Локальная переменная
   {Address<4>.AddrMode=Index;
    if (Val<2>=Level<Block>) // Переменная текущего уровня
      {Address<4>.AddrReg=A6;
       Address<4>.AddrDisp=Table[Val<3>];
      }
    else
      {Address<4>.AddrMode=IndPost;
       Address<4>.AddrReg=N0;
       Address<4>.IndexReg=N0;
       Address<4>.AddrDisp=Display[Val<2>];
       Address<4>.IndexDisp=Table[Val<3>];
      }
   }.
```

Рассмотрим трансляцию доступа к полям записи. Она описывается следующим правилом (Number – это Лидер-номер описания поля):

```
RULE
VarTail ::= 'FIL' Number VarTail
SEMANTICS
if (Address<0>.AddrMode==Abs)
  {Address<3>.AddrMode=Abs;
   Address<3>.AddrDisp=Address<0>.AddrDisp+Table[Val<2>];
  }
else
  {Address<3>=Address<0>;
   if (Address<0>.AddrMode==Index)
     Address<3>.AddrDisp=Address<0>.AddrDisp+Table[Val<2>];
   else
     Address<3>.IndexDisp=Address<0>.IndexDisp+Table[Val<2>];
  }.
```

9.5 Трансляция целых выражений

Трансляция выражений различных типов управляется синтаксически благодаря наличию указателя типа перед каждой операцией. Мы рассмотрим некоторые наиболее характерные проблемы генерации кода для выражений.

Система команд MC68020 обладает двумя особенностями, сказывающимися на генерации кода для арифметических выражений (то же можно сказать и о генерации кода для выражений типа “множества”):

1) один из операндов выражения (правый) должен при выполнении операции находиться на регистре, поэтому если оба операнда не на ре-

гистрах, то перед выполнением операции один из них надо загрузить на регистр;

2) система команд довольно “симметрична”, т.е. нет специальных требований к регистрам при выполнении операций (таких, например, как пары регистров или требования четности и т.д.).

Поэтому выбор команд при генерации арифметических выражений определяется довольно простыми таблицами решений. Например, для целочисленного сложения такая таблица приведена на рис. 9.4.

		Правый операнд A2	
		R	V
Левый операнд A1	R	ADD A1,A2	ADD A2, A1
	V	ADD A1, A2	MOVE A1, R ADD A2, R

Рис. 9.4:

Здесь имеется в виду, что R – операнд на регистре, V – переменная или константа. Такая таблица решений должна также учитывать коммутативность операций.

```
RULE
IntExpr ::= 'PLUS' IntExpr IntExpr
SEMANTICS
if (Address<2>.AddrMode!=D) && (Address<3>.AddrMode!=D)
  {Address<0>.AddrMode=D;
   Address<0>.Addreg=GetFree(RegSet<Block>);
   Emit2(MOVE,Address<2>,Address<0>);
   Emit2(ADD,Address<2>,Address<0>);
  }
else
  if (Address<2>.AddrMode==D)
    {Emit2(ADD,Address<3>,Address<2>);
     Address<0>:=Address<2>);
    }
  else {Emit2(ADD,Address<2>,Address<3>);
       Address<0>:=Address<3>);
      }.
```

9.6 Трансляция арифметических выражений

Одной из важнейших задач при генерации кода является распределение регистров. Рассмотрим хорошо известную технику распределения ре-

гистров при трансляции арифметических выражений, называемую алгоритмом Сети-Ульмана. (*Замечание*: в целях большей наглядности, в данном параграфе мы немного отступаем от семантики арифметических команд MC68020 и предполагаем, что команда

```
Op Arg1, Arg2
```

выполняет действие `Arg2:=Arg1 Op Arg2`.)

Пусть система команд машины имеет неограниченное число универсальных регистров, в которых выполняются арифметические команды. Рассмотрим, как можно сгенерировать код, используя для данного арифметического выражения минимальное число регистров.

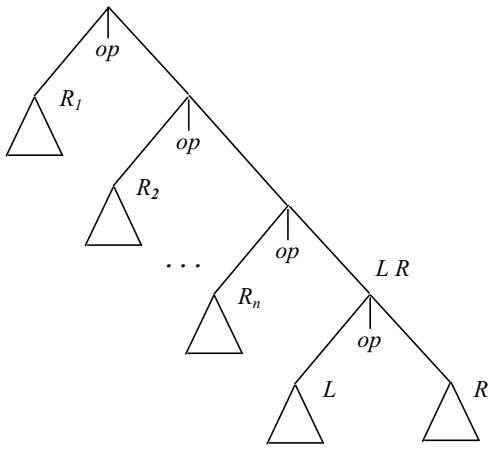


Рис. 9.5:

Пусть имеется синтаксическое дерево выражения. Предположим сначала, что распределение регистров осуществляется по простейшей схеме сверху-вниз слева-направо, как изображено на рис. 9.5. Тогда к моменту генерации кода для поддерева LR занято n регистров. Пусть поддерево L требует n_l регистров, а поддерево $R - n_r$ регистров. Если $n_l = n_r$, то при вычислении L будет использовано n_l регистров и под результат будет занят $(n+1)$-й регистр. Еще $n_r (= n_l)$ регистров будет использовано при вычислении R. Таким образом, общее число использованных регистров будет равно $n + n_l + 1$.

Если $n_l > n_r$, то при вычислении L будет использовано n_l регистров. При вычислении R будет использовано $n_r < n_l$ регистров, и всего будет использовано не более чем $n + n_l$ регистров. Если $n_l < n_r$, то после вычисления L под результат будет занят один регистр (предположим, $(n+1)$-й) и n_r регистров будет использовано для вычисления R. Всего будет использовано $n + n_r + 1$ регистров.

Видно, что для деревьев, совпадающих с точностью до порядка потомков каждой вершины, минимальное число регистров при распреде-

лении их слева-направо достигается на дереве, у которого в каждой вершине слева расположено более “сложное” поддерево, требующее большего числа регистров. Таким образом, если дерево таково, что в каждой внутренней вершине правое поддерево требует меньшего числа регистров, чем левое, то, обходя дерево слева направо, можно оптимально распределить регистры. Без перестройки дерева это означает, что если в некоторой вершине дерева справа расположено более сложное поддерево, то сначала сгенерируем код для него, а затем уже для левого поддерева.

Алгоритм работает следующим образом. Сначала осуществляется разметка синтаксического дерева по следующим правилам.

Правила разметки:

1) если вершина – правый лист или дерево состоит из единственной вершины, помечаем эту вершину числом 1, если вершина – левый лист, помечаем ее 0 (рис. 9.6).

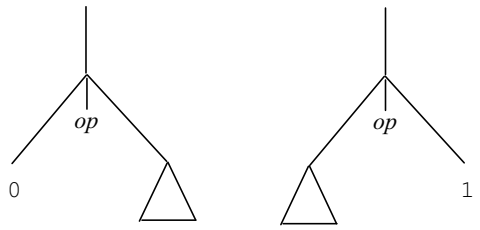


Рис. 9.6:

2) если вершина имеет прямых потомков с метками l_1 и l_2, то в качестве метки этой вершины выбираем наибольшее из чисел l_1 или l_2 либо число $l_1 + 1$, если $l_1 = l_2$ (рис. 9.7).

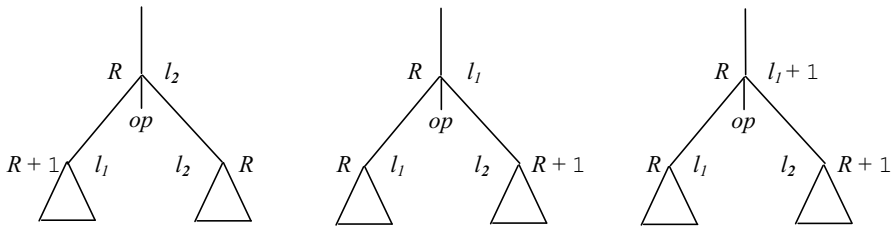


Рис. 9.7:

Эта разметка позволяет определить, какое из поддеревьев требует большего количества регистров для своего вычисления. Далее осуществляется распределение регистров для результатов операций по следующим правилам:

1) Корню назначается первый регистр.

2) Если метка левого потомка меньше метки правого, то левому потомку назначается регистр на единицу больший, чем предку, а правому – с тем же номером (сначала вычисляется правое поддерево и его результат помещается в регистр R), так что регистры занимаются последовательно. Если же метка левого потомка больше или равна метке правого потомка, то наоборот, правому потомку назначается регистр на единицу больший, чем предку, а левому – с тем же номером (сначала вычисляется левое поддерево и его результат помещается в регистр R – рис. 9.7).

После этого формируется код по следующим правилам:

1) если вершина – правый лист с меткой 1, то ей соответствует код

```
MOVE X, R
```

где R – регистр, назначенный этой вершине, а X – адрес переменной, связанной с вершиной (рис. 9.8, б);

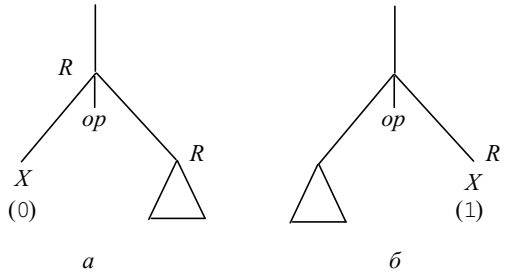


Рис. 9.8:

2) если вершина внутренняя и ее левый потомок – лист с меткой 0, то ей соответствует код

```
Код правого поддерева
Op X, R
```

где R – регистр, назначенный этой вершине, X – адрес переменной, связанной с вершиной, а Op – операция, примененная в вершине (рис. 9.8, а);

3) если непосредственные потомки вершины не листья и метка правой вершины больше метки левой, то вершине соответствует код

```
Код правого поддерева
Код левого поддерева
Op R+1, R
```

где R – регистр, назначенный внутренней вершине, и операция Op, вообще говоря, не коммутативная (рис. 9.9, б);

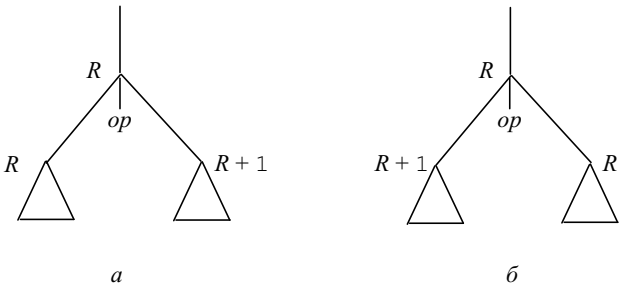


Рис. 9.9:

4) если непосредственные потомки вершины не листья и метка правой вершины меньше или равна метке левой вершины, то вершине соответствует код

```
Код левого поддерева
Код правого поддерева
Op R, R+1
MOVE R+1, R
```

Последняя команда генерируется для того, чтобы получить результат в нужном регистре (в случае коммутативной операции операнды операции можно поменять местами и избежать дополнительной пересылки – рис. 9.9, а).

Рассмотрим атрибутную схему, реализующую эти правила генерации кода (для большей наглядности входная грамматика соответствует обычной инфиксной записи, а не Лидер-представлению). В этой схеме генерация кода происходит не непосредственно в процессе обхода дерева, как раньше, а из-за необходимости переставлять поддеревья код строится в виде текста с помощью операции конкатенации. Практически, конечно, это нецелесообразно: разумнее управлять обходом дерева непосредственно, однако для простоты мы будем пользоваться конкатенацией.

```
RULE
Expr ::= IntExpr
SEMANTICS
Reg<1>=1; Code<0>=Code<1>; Left<1>=true.

RULE
IntExpr ::= IntExpr Op IntExpr
```

```
SEMANTICS
Left<1>=true; Left<3>=false;
Label<0>=(Label<1>==Label<3>)
       ? Label<1>+1
       : Max(Label<1>,Label<3>);
Reg<1>=(Label<1> < Label<3>)
     ? Reg<0>+1
     : Reg<0>;
Reg<3>=(Label<1> < Label<3>)
     ? Reg<0>
     : Reg<0>+1;
Code<0>=(Label<1>==0)
      ? Code<3> + Code<2>
        + Code<1> + "," + Reg<0>
      : (Label<1> < Label<3>)
        ? Code<3> + Code<1> + Code<2> +
          (Reg<0>+1) + "," + Reg<0>
        : Code<1> + Code<3> + Code<2> +
          Reg<0> + "," + (Reg<0>+1)
          + "MOVE" + (Reg<0>+1) + "," + Reg<0>.

RULE
IntExpr ::= Ident
SEMANTICS
Label<0>=(Left<0>) ? 0 : 1;
Code<0>=(!Left<0>)
      ? "MOVE" + Reg<0> + "," + Val<1>
      : Val<1>.

RULE
IntExpr ::= '(' IntExpr ')'
SEMANTICS
Label<0>=Label<2>;
Reg<2>=Reg<0>;
Code<0>=Code<2>.

RULE
Op ::= '+'
SEMANTICS
Code<0>="ADD".

RULE
Op ::= '-'
SEMANTICS
Code<0>="SUB".
```

```
RULE
Op ::='*'
SEMANTICS
Code<0>="MUL".

RULE
Op ::='/'
SEMANTICS
Code<0>="DIV".
```

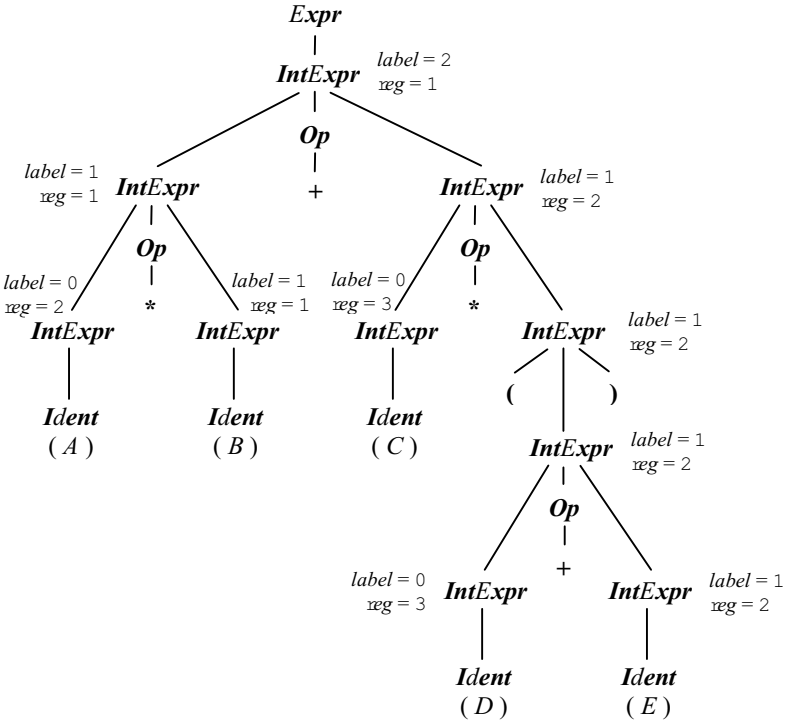


Рис. 9.10:

Атрибутированное дерево для выражения A*B+C*(D+E) приведено на рис. 9.10. При этом будет сгенерирован следующий код:

```
MOVE B, R1
MUL  A, R1
MOVE E, R2
ADD  D, R2
MUL  C, R2
```

```
ADD  R1, R2
MOVE R2, R1
```

Приведенная атрибутная схема требует двух проходов по дереву выражения. Рассмотрим теперь другую атрибутную схему, в которой достаточно одного обхода для генерация программы для выражений с оптимальным распределением регистров [7].

Пусть мы произвели разметку дерева разбора так же, как и в предыдущем алгоритме. Назначение регистров будем производить следующим образом.

Левому потомку всегда назначается регистр, равный его метке, а правому – его метке, если она не равна метке его левого брата, и метке + 1, если метки равны. Поскольку более сложное поддерево всегда вычисляется раньше более простого, его регистр результата имеет больший номер, чем любой регистр, используемый при вычислении более простого поддерева, что гарантирует правильность использования регистров.

Приведенные соображения реализуются следующей атрибутной схемой:

```
RULE
Expr ::= IntExpr
SEMANTICS
Code<0>=Code<1>; Left<1>=true.

RULE
IntExpr ::= IntExpr Op IntExpr
SEMANTICS
Left<1>=true; Left<3>=false;
Label<0>=(Label<1>==Label<3>)
      ? Label<1>+1
      : Max(Label<1>,Label<3>);
Code<0>=(Label<3> > Label<1>)
     ? (Label<1>==0)
       ? Code<3> + Code<2> + Code<1>
        + "," + Label<3>
       : Code<3> + Code<1> + Code<2> +
        Label<1> + "," + Label<3>
     : (Label<3> < Label<1>)
       ? Code<1> + Code<3> + Code<2> +
        Label<1> + "," + Label<3> +
        "MOVE" + Label<3> + "," + Label<1>
       : // метки равны
        Code<3> + "MOVE" + Label<3> +
        "," + Label<3>+1 + Code<1> +
        Code<2> + Label<1> + "," +
        Label<1>+1.
```

```
RULE
IntExpr ::= Ident
SEMANTICS
Label<0>=(Left<0>) ? 0 : 1;
Code<0>=(Left<0>) ? Val<1>
     : "MOVE" + Val<1> + "R1".

RULE
IntExpr ::= '(' IntExpr ')'
SEMANTICS
Label<0>=Label<2>;
Code<0>=Code<2>.

RULE
Op ::= '+'
SEMANTICS
Code<0>="ADD".

RULE
Op ::= '-'
SEMANTICS
Code<0>="SUB".

RULE
Op ::= '*'
SEMANTICS
Code<0>="MUL".

RULE
Op ::= '/'
SEMANTICS
Code<0>="DIV".
```

Команды пересылки требуются для согласования номеров регистров, в которых осуществляется выполнение операции, с регистрами, в которых должен быть выдан результат. Это имеет смысл, когда эти регистры разные. Получиться это может из-за того, что по приведенной схеме результат выполнения операции всегда находится в регистре с номером метки, а метки левого и правого поддеревьев могут совпадать.

Для выражения A*B+C*(D+E) будет сгенерирован следующий код:

```
MOVE E, R1
ADD  D, R1
MUL  C, R1
MOVE R1, R2
```

```
MOVE B, R1
MUL  A, R1
ADD  R1, R2
```

В приведенных атрибутных схемах предполагалось, что регистров достаточно для трансляции любого выражения. Если это не так, приходится усложнять схему трансляции и при необходимости сбрасывать содержимое регистров в память (или магазин).

9.7 Трансляция логических выражений

Логические выражения, включающие логическое умножение, логическое сложение и отрицание, можно вычислять как непосредственно, используя таблицы истинности, так и с помощью условных выражений, основанных на следующих простых правилах:

A AND B эквивалентно if A then B else False,
A OR B эквивалентно if A then True else B.

Если в качестве компонент выражений могут входить функции с побочным эффектом, то, вообще говоря, результат вычисления может зависеть от способа вычисления. В некоторых языках программирования не оговаривается, каким способом должны вычисляться логические выражения (например, в Паскале), в некоторых требуется, чтобы вычисления производились тем или иным способом (например, в Модуле-2 требуется, чтобы выражения вычислялись по приведенным формулам), в некоторых языках есть возможность явно задать способ вычисления (Си, Ада). Вычисление логических выражений непосредственно по таблицам истинности аналогично вычислению арифметических выражений, поэтому мы не будем их рассматривать отдельно. Рассмотрим подробнее способ вычисления с помощью приведенных выше формул (будем называть его “вычислением с условными переходами”). Иногда такой способ рассматривают как оптимизацию вычисления логических выражений.

Рассмотрим следующую атрибутную грамматику со входным языком логических выражений:

```
RULE
Expr ::= BoolExpr
SEMANTICS
FalseLab<1>=False; TrueLab<1>=True;
Code<0>=Code<1>.

RULE
BoolExpr ::= BoolExpr 'AND' BoolExpr
SEMANTICS
FalseLab<1>=FalseLab<0>; TrueLab<1>=NodeLab<3>;
FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>;
```

```
Code<0>=NodeLab<0> + ":" + Code<1> + Code<3>.

RULE
BoolExpr ::= BoolExpr 'OR' BoolExpr
SEMANTICS
FalseLab<1>=NodeLab<3>; TrueLab<1>=TrueLab<0>;
FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>;
Code<0>=NodeLab<0> + ":" + Code<1> + Code<3>.

RULE
BoolExpr ::= '(' BoolExpr ')'
SEMANTICS
FalseLab<2>=FalseLab<0>;
TrueLab<2>=TrueLab<0>;
Code<0>=NodeLab<0> + ":" + Code<2>.

RULE
BoolExpr ::= F
SEMANTICS
Code<0>=NodeLab<0> + ":" + "GOTO" + FalseLab<0>.

RULE
BoolExpr ::= T
SEMANTICS
Code<0>=NodeLab<0> + ":" + "GOTO" + TrueLab<0>.
```

Здесь предполагается, что все вершины дерева занумерованы и номер вершины дает атрибут NodeLab. Метки вершин передаются, как это изображено на рис. 9.11.

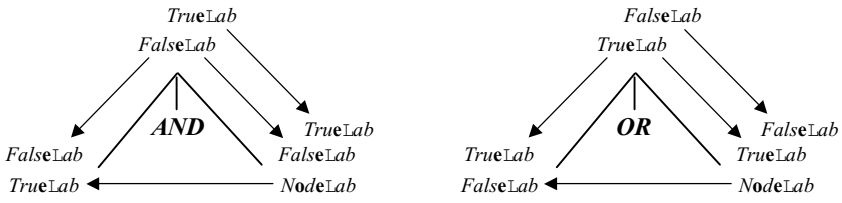


Рис. 9.11:

Таким образом, каждому атрибутированному дереву в этой атрибутной грамматике сопоставляется код, полученный в результате обхода

дерева сверху-вниз слева-направо следующим образом. При входе в вершину BoolExpr генерируется ее номер, в вершине F генерируется текст GOTO *значение атрибута* FalseLab<0>, в вершине T – GOTO *значение атрибута* TrueLab<0>. Например, для выражения

```
F OR ( F AND T AND T ) OR T
```

получим атрибутированное дерево, изображенное на рис. 9.12, и код

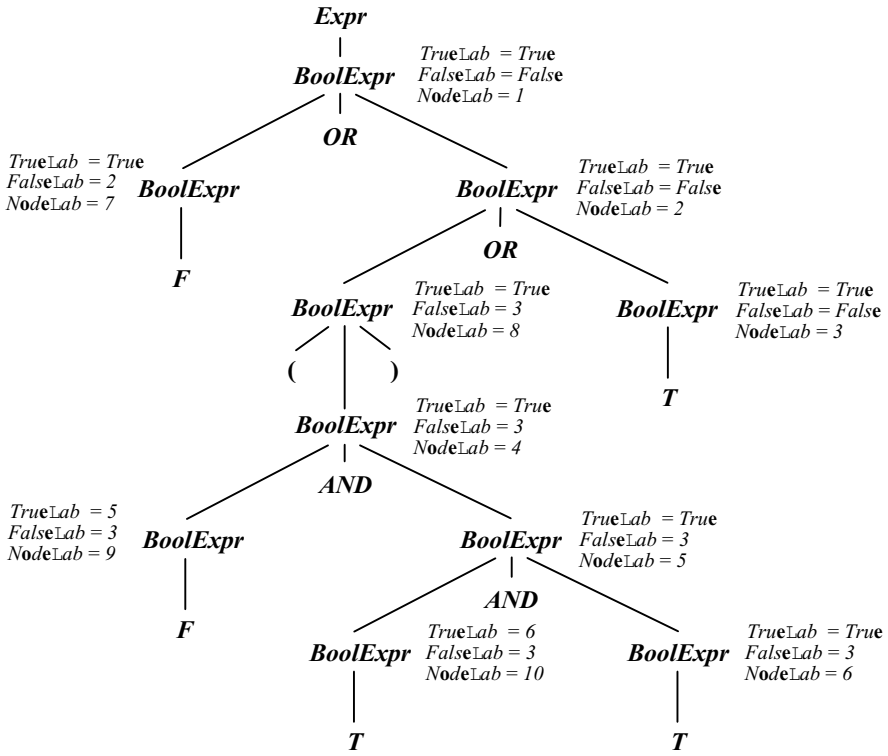


Рис. 9.12:

```
1:7:    GOTO 2
2:8:4:9: GOTO 3
5:10:   GOTO 6
6:      GOTO True
3:      GOTO True
True: ...
False: ...
```

Эту линеаризованную запись можно трактовать как программу вычисления логического значения: каждая строка может быть помечена

номером вершины и содержать либо переход на другую строку, либо переход на True или False, что соответствует значению выражения true или false. Будем говорить, что полученная программа *вычисляет* (или *интерпретирует*) значение выражения, если в результате ее выполнения (от первой строки) мы придем к строке, содержащей GOTO True или GOTO False.

Утверждение 9.1. *В результате интерпретации поддерева с некоторыми значениями атрибутов* FalseLab *и* TrueLab *в его корне выполняется команда* GOTO TrueLab, *если значение выражения истинно, и команда* GOTO FalseLab, *если значение выражения ложно.*

Доказательство. Применим индукцию по высоте дерева. Для деревьев высоты 1, соответствующих правилам

BoolExpr ::= F и BoolExpr ::= T,

справедливость утверждения следует из соответствующих атрибутных правил. Пусть дерево имеет высоту $n > 1$. Зависимость атрибутов для дизъюнкции и конъюнкции приведена на рис. 9.13.

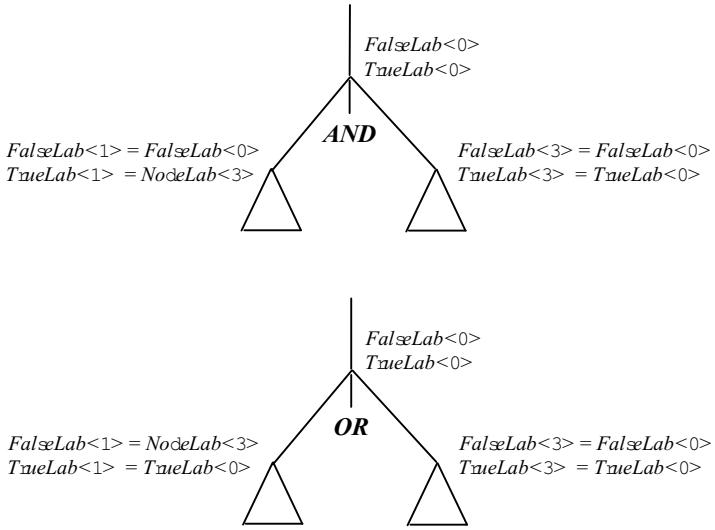


Рис. 9.13:

Если для конъюнкции значение левого поддерева ложно и по индукции вычисление левого поддерева завершается командой GOTO FalseLab<1>, то получаем, что вычисление всего дерева завершается командой перехода GOTO FalseLab<0> (= FalseLab<1>). Если же значение левого поддерева истинно, то его вычисление завершается командой перехода GOTO TrueLab<1> (= NodeLab<3>). Если значение правого поддерева ложно, то вычисление

всего дерева завершается командой GOTO FalseLab<0> (= FalseLab<3>). Если же оно истинно, вычисление всего дерева завершается командой перехода GOTO TrueLab<0> (= TrueLab<3>). Аналогично – для дизъюнкции. ■

Утверждение 9.2. *Для любого логического выражения, состоящего из констант, программа, полученная в результате обхода дерева этого выражения, завершается со значением логического выражения в обычной интерпретации, т.е. осуществляется переход на* True *для значения, равного* $true$*, и переход на метку* False *для значения* $false$.

Доказательство. Это утверждение является частным случаем предыдущего. Его справедливость следует из того, что метки корня дерева равны соответственно TrueLab = True и FalseLab = False. ■

Добавим теперь новое правило в предыдущую грамматику:

```
RULE
BoolExpr ::= Ident
SEMANTICS
Code<0>=NodeLab<0> + ":" + "if (" + Val<1> + "==T) GOTO" +
     TrueLab<0> + "else GOTO" + FalseLab<0>.
```

Тогда, например, для выражения A OR (B AND C AND D) OR E получим следующую программу:

```
1:7:    if (A==T) GOTO True else GOTO 2
2:8:4:9: if (B==T) GOTO 5 else GOTO 3
5:10:   if (C==T) GOTO 6 else GOTO 3
6:      if (D==T) GOTO True else GOTO 3
3:      if (E==T) GOTO True else GOTO False
True:  ...
False: ...
```

При каждом конкретном наборе данных эта программа превращается в программу вычисления логического значения.

Утверждение 9.3. *В каждой строке программы, сформированной предыдущей атрибутной схемой, одна из меток внутри условного оператора совпадает с меткой следующей строки.*

Доказательство. Действительно, по правилам наследования атрибутов TrueLab и FalseLab, в правилах для дизъюнкции и конъюнкции либо атрибут FalseLab, либо атрибут TrueLab принимает значение метки следующего поддерева. Кроме того, как значение FalseLab, так и значение TrueLab, передаются в правое поддерево от предка. Таким образом, самый правый потомок всегда имеет одну из меток TrueLab или FalseLab, равную метке правого брата соответствующего поддерева. Учитывая порядок генерации команд, получаем справедливость утверждения. ■

Дополним теперь атрибутную грамматику следующим образом:

```
RULE
Expr ::= BoolExpr
SEMANTICS
FalseLab<1>=False; TrueLab<1>=True;
Sign<1>=false;
Code<0>=Code<1>.

RULE
BoolExpr ::= BoolExpr 'AND' BoolExpr
SEMANTICS
FalseLab<1>=FalseLab<0>; TrueLab<1>=NodeLab<3>;
FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>;
Sign<1>=false; Sign<3>=Sign<0>;
Code<0>=NodeLab<0> + ":" + Code<1> + Code<3>.

RULE
BoolExpr ::= BoolExpr 'OR' BoolExpr
SEMANTICS
FalseLab<1>=NodeLab<3>; TrueLab<1>=TrueLab<0>;
FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>;
Sign<1>=true; Sign<3>=Sign<0>;
Code<0>=NodeLab<0> + ":" + Code<1> + Code<3>.

RULE
BoolExpr ::= 'NOT' BoolExpr
SEMANTICS
FalseLab<2>=TrueLab<0>; TrueLab<2>=FalseLab<0>;
Sign<2>=! Sign<0>;
Code<0>=Code<2>.

RULE
BoolExpr ::= '(' BoolExpr ')'
SEMANTICS
FalseLab<2>=FalseLab<0>;
TrueLab<2>=TrueLab<0>;
Sign<2>=Sign<0>;
Code<0>=NodeLab<0> + ":" + Code<2>.

RULE
BoolExpr ::= F
SEMANTICS
Code<0>=NodeLab<0> + ":" + "GOTO" + FalseLab<0>.

RULE
```

```
BoolExpr ::= T
SEMANTICS
Code<0>=NodeLab<0> + ":" + "GOTO" + TrueLab<0>.

RULE
BoolExpr ::= Ident
SEMANTICS
Code<0>=NodeLab<0> + ":" + "if (" + Val<1> + "==T) GOTO"
     + TrueLab<0> + "else GOTO" + FalseLab<0>.
```

Правила наследования атрибута Sign приведены на рис. 9.14.

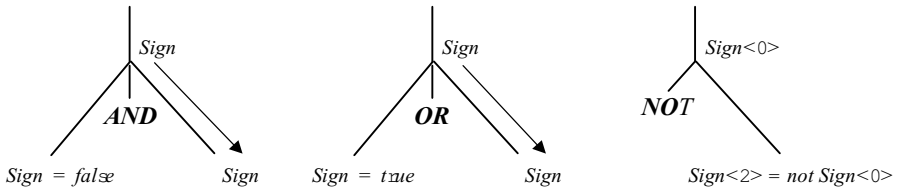


Рис. 9.14:

Программу желательно сформировать таким образом, чтобы else-метка была как раз меткой следующей вершины. Как это можно сделать, следует из следующего утверждения.

Утверждение 9.4. *В каждой терминальной вершине, метка ближайшего правого для нее поддерева равна значению атрибута* FalseLab *этой вершины, тогда и только тогда, когда значение атрибута* Sign *этой вершины равно* $true$*, и наоборот, метка ближайшего правого для нее поддерева равна значению атрибута* TrueLab *этой вершины, тогда и только тогда, когда значение атрибута* Sign *равно* $false$*.*

Доказательство. Действительно, если ближайшей общей вершиной является AND, то в левого потомка передается NodeLab правого потомка в качестве TrueLab и одновременно Sign правого потомка равен $true$. Если же ближайшей общей вершиной является OR, то в левого потомка передается NodeLab правого потомка в качестве FalseLab и одновременно Sign правого потомка равен $false$. Во все же правые потомки значения TrueLab, FalseLab и Sign передаются из предка (за исключением правила для NOT, в котором TrueLab и FalseLab меняются местами, но одновременно на противоположный меняется и Sign). ■

Эти два утверждения (3 и 4) позволяют заменить последнее правило атрибутной грамматики следующим образом:

```
RULE
BoolExpr ::= Ident
SEMANTICS
Code<0>=NodeLab<0> + ":" +
     (Sign<0>
      ? "if (" + Val<1> + "==T) GOTO" + TrueLab<0>
      : "if (" + Val<1> + "==F) GOTO" + FalseLab<0>).
```

В свою очередь, при генерации машинных команд это правило можно заменить на следующее:

```
RULE
BoolExpr ::= Ident
SEMANTICS
Code<0>=NodeLab<0> + ":" + "TST" + Val<1> +
     (Sign<0>
      ? "BNE" + TrueLab<0>
      : "BEQ" + FalseLab<0>).
```

Таким образом, для выражения `A OR ( B AND C AND D ) OR E` получим следующий код на командах перехода:

```
 1:7:    TST A
         BNE True
 2:8:4:9: TST B
         BEQ 3
 5:10:   TST C
         BEQ 3
 6:       TST D
         BNE True
 3:       TST E
         BEQ False
 True:  ...
 False: ...
```

Если элементом логического выражения является сравнение, то генерируется команда, соответствующая знаку сравнения (`BEQ` для `=`, `BNE` для `<>`, `BGE` для `>=` и т.д.), если атрибут Sign соответствующей вершины имеет значение $true$, и отрицание (`BNE` для `=`, `BEQ` для `<>`, `BLT` для `>=` и т.д.), если атрибут Sign имеет значение $false$.

9.8 Выделение общих подвыражений

Выделение общих подвыражений относится к области оптимизации программ. В общем случае трудно (или даже невозможно) провести грани-

цу между оптимизацией и “качественной трансляцией”. Оптимизация - это и есть качественная трансляция. Обычно термин “оптимизация” употребляют, когда для повышения качества программы используют ее глубокие преобразования такие, например, как перевод в графовую форму для изучения нетривиальных свойств программы.

В этом смысле выделение общих подвыражений – одна из простейших оптимизаций. Для ее осуществления требуется некоторое преобразование программы, а именно построение ориентированного ациклического графа, о котором говорилось в главе, посвященной промежуточным представлениям.

Линейный участок – это последовательность операторов, в которую управление входит в начале и выходит в конце без остановки и перехода изнутри.

Рассмотрим дерево линейного участка, в котором вершинами служат операции, а потомками – операнды. Будем говорить, что две вершины образуют *общее подвыражение*, если поддеревья для них совпадают, т.е. имеют одинаковую структуру и, соответственно, одинаковые операции во внутренних вершинах и одинаковые операнды в листьях. Выделение общих подвыражений позволяет генерировать для них код один раз, хотя может привести и к некоторым трудностям, о чем вкратце будет сказано ниже.

Выделение общих подвыражений проводится на линейном участке и основывается на двух положениях.

1. Поскольку на линейном участке переменной может быть несколько присваиваний, то при выделении общих подвыражений необходимо различать вхождения переменных до и после присваивания. Для этого каждая переменная снабжается счетчиком. Вначале счетчики всех переменных устанавливаются равными 0. При каждом присваивании переменной ее счетчик увеличивается на 1.

2. Выделение общих подвыражений осуществляется при обходе дерева выражения снизу вверх слева направо. При достижении очередной вершины (пусть операция, примененная в этой вершине, есть бинарная *op*; в случае унарной операции рассуждения те же) просматриваем общие подвыражения, связанные с *op*. Если имеется выражение, связанное с *op* и такое, что его левый операнд есть общее подвыражение с левым операндом нового выражения, а правый операнд - общее подвыражение с правым операндом нового выражения, то объявляем новое выражение общим с найденным и в новом выражении запоминаем указатель на найденное общее выражение. Базисом построения служит переменная: если операндами обоих выражений являются одинаковые переменные с одинаковыми счетчиками, то они являются общими подвыражениями. Если выражение не выделено как общее, оно заносится в список операций, связанных с *op*.

Рассмотрим теперь реализацию алгоритма выделения общих подвыражений. Поддерживаются следующие глобальные переменные:

Table – таблица переменных; для каждой переменной хранится ее счет-

чик (Count) и указатель на вершину дерева выражений, в которой переменная встретилась в последний раз в правой части (Last);

OpTable – таблица списков (типа LisType) общих подвыражений, связанных с каждой операцией. Каждый элемент списка хранит указатель на вершину дерева (поле Addr) и продолжение списка (поле List).

С каждой вершиной дерева выражения связана запись типа NodeType, со следующими полями:

Left – левый потомок вершины,

Right – правый потомок вершины,

Comm – указатель на предыдущее общее подвыражение,

Flag – признак, является ли поддерево общим подвыражением,

Varbl – признак, является ли вершина переменной,

VarCount – счетчик переменной.

Выделение общих подвыражений и построение дерева осуществляются приведенными ниже правилами. Атрибут Entry нетерминала Variable дает указатель на переменную в таблице Table. Атрибут Val символа Op дает код операции. Атрибут Node символов IntExpr и Assignment дает указатель на запись типа NodeType соответствующего нетерминала.

```
RULE
Assignment ::= Variable IntExpr
SEMANTICS
Table[Entry<1>].Count=Table[Entry<1>].Count+1.
// Увеличить счетчик присваиваний переменной

RULE
IntExpr ::= Variable
SEMANTICS
if ((Table[Entry<1>].Last!=NULL)
   // Переменная уже была использована
   && (Table[Entry<1>].Last->VarCount
         == Table[Entry<1>].Count ))
   // С тех пор переменной не было присваивания
  {Node<0>->Flag=true;
   // Переменная - общее подвыражение
   Node<0>->Comm= Table[Entry<1>].Last;
   // Указатель на общее подвыражение
  }
else Node<0>->Flag=false;

Table[Entry<1>].Last=Node<0>;
// Указатель на последнее использование переменной
Node<0>->VarCount= Table[Entry<1>].Count;
// Номер использования переменной
Node<0>->Varbl=true.
// Выражение - переменная
```

```
RULE
IntExpr ::= Op IntExpr IntExpr
SEMANTICS
LisType * L; //Тип списков операции
if ((Node<2>->Flag) && (Node<3>->Flag))
  // И справа, и слева - общие подвыражения
  {L=OpTable[Val<1>];
  // Начало списка общих подвыражений для операции
   while (L!=NULL)
    if ((Node<2>==L->Left)
       && (Node<3>==L->Right))
         // Левое и правое поддеревья совпадают
     break;
    else L=L->List;// Следующий элемент списка
  }
else L=NULL; //Не общее подвыражение

Node<0>->Varbl=false; // Не переменная
Node<0>->Comm=L;
//Указатель на предыдущее общее подвыражение или NULL

if (L!=NULL)
   {Node<0>->Flag=true; //Общее подвыражение
    Node<0>->Left=Node<2>;
    // Указатель на левое поддерево
    Node<0>->Right=Node<3>;
    // Указатель на правое поддерево
   }
else {Node<0>->Flag=false;
      // Данное выражение не может рассматриваться как общее
      // Если общего подвыражения с данным не было,
      // включить данное в список для операции
      L=alloc(sizeof(struct LisType));
      L->Addr=Node<0>;
      L->List=OpTable[Val<1>];
      OpTable[Val<1>]=L;
     }.
```

Рассмотрим теперь некоторые простые правила распределения регистров при наличии общих подвыражений. Если число регистров ограничено, можно выбрать один из следующих двух вариантов.

1. При обнаружении общего подвыражения с подвыражением в уже просмотренной части дерева (и, значит, с уже распределенными регистрами) проверяем, расположено ли его значение на регистре. Если да, и если регистр после этого не менялся, заменяем вычисление поддерева

на значение в регистре. Если регистр менялся, то вычисляем подвыражение заново.

2. Вводим еще один проход. На первом проходе распределяем регистры. Если в некоторой вершине обнаруживается, что ее поддерево общее с уже вычисленным ранее, но значение регистра потеряно, то в такой вершине на втором проходе необходимо сгенерировать команду сброса регистра в рабочую память. Выигрыш в коде будет, если стоимость команды сброса регистра + доступ к памяти в повторном использовании этой памяти не превосходит стоимости заменяемого поддерева. Поскольку стоимость команды MOVE известна, можно сравнить стоимости и принять оптимальное решение: пометить предыдущую вершину для сброса либо вычислять поддерево полностью.

9.9 Генерация оптимального кода методами синтаксического анализа

9.9.1 Сопоставление образцов

Техника генерации кода, рассмотренная выше, основывалась на однозначном соответствии структуры промежуточного представления и описывающей это представление грамматики. Недостатком такого "жесткого" подхода является то, что как правило одну и ту же программу на промежуточном языке можно реализовать многими различными способами в системе команд машины. Эти разные реализации могут иметь различную длину, время выполнения и другие характеристики. Для генерации более качественного кода может быть применен подход, изложенный в настоящей главе.

Этот подход основан на понятии "сопоставления образцов": командам машины сопоставляются некоторые "образцы", вхождения которых ищутся в промежуточном представлении программы, и делается попытка "покрыть" промежуточную программу такими образцами. Если это удается, то по образцам восстанавливается программа уже в кодах. Каждое такое покрытие соответствует некоторой программе, реализующей одно и то же промежуточное представление.

На рис. 9.15 показано промежуточное дерево для оператора $a = b[i] + 5$, где a, b, i – локальные переменные, хранимые со смещениями x, y, z соответственно в областях данных с одноименными адресами.

Элемент массива b занимает память в одну машинную единицу. 0-местная операция $const$ возвращает значение атрибута соответствующей вершины промежуточного дерева, указанного на рисунке в скобках после оператора. Одноместная операция @ означает косвенную адресацию и возвращает содержимое регистра или ячейки памяти, имеющей адрес, задаваемый аргументом операции.

На рис. 9.16 показан пример сопоставления образцов машинным командам. Приведены два варианта задания образца: в виде дерева и в

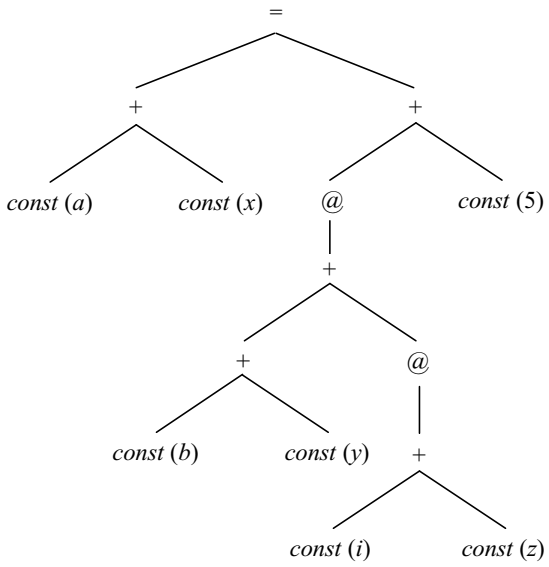


Рис. 9.15:

виде правила контекстно-свободной грамматики. Для каждого образца указана машинная команда, реализующая этот образец, и стоимость этой команды.

В каждом дереве-образце корень или лист может быть помечен терминальным и/или нетерминальным символом. Внутренние вершины помечены терминальными символами – знаками операций. При наложении образца на дерево выражения, во-первых, терминальный символ образца должен соответствовать терминальному символу дерева, и, во-вторых, образцы должны “склеиваться” по типу нетерминального символа, т.е. тип корня образца должен совпадать с типом вершины, в которую образец подставляется корнем. Допускается использование “цепных” образцов, т.е. образцов, корню которых не соответствует терминальный символ, и имеющих единственный элемент в правой части. Цепные правила служат для приведения вершин к одному типу. Например, в рассматриваемой системе команд одни и те же регистры используются как для целей адресации, так и для вычислений. Если бы в системе команд для этих целей использовались разные группы регистров, то в грамматике команд могли бы использоваться разные нетерминалы, а для пересылки из адресного регистра в регистр данных могла бы использоваться соответствующая команда и образец.

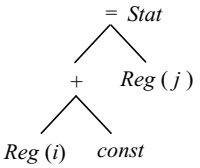
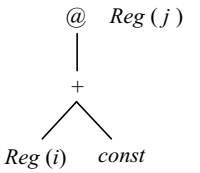
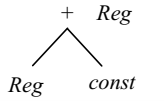
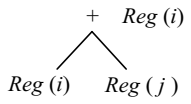
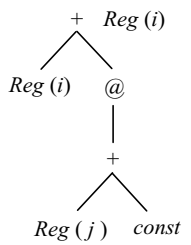
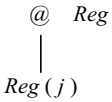
№	Образец	Правило грамматики	Команда / стоимость	
1		*Reg* → *const*	`MOVE #const, R`	2
2		*Stat* → '=' '+' *Reg const Reg*	`MOVE Rj, const(Ri)`	4
3		*Reg* → '@' +' *Reg const*	`MOVE const(Ri), Rj`	4
4		*Reg* → '+' *Reg const*	`ADD #const,R`	3
5		*Reg* → '+' *Reg Reg*	`ADD Rj,Ri`	2
6		*Reg* → '+' *Reg* '@' '+' *Reg const*	`ADD const(Rj),Ri`	4
7		*Reg* → '@' *Reg*	`MOVE (Rj),Ri`	2

Рис. 9.16:

Нетерминалы Reg на образцах могут быть помечены индексом (i или j), что (неформально) соответствует номеру регистра и служит лишь для пояснения смысла использования регистров. Отметим, что при генерации кода рассматриваемым методом не осуществляется распределение регистров. Это является отдельной задачей.

Стоимость может определяться различными способами, например чис-

лом обращений к памяти при выборке и исполнении команды. Здесь мы не рассматриваем этого вопроса. На рис. 9.17 приведен пример покрытия промежуточного дерева рис. 9.15 образцами рис. 9.16. В рамки заключены фрагменты дерева, сопоставленные образцу правила, номер которого указывается в левом верхнем углу рамки. В квадратных скобках указаны результирующие вершины.

Приведенное покрытие дает такую последовательность команд:

```
MOVE b,Rb
ADD  #y,Rb
MOVE i,Ri
ADD  z(Ri),Rb
MOVE (Rb),Rb
ADD  #5,Rb
MOVE a,Ra
MOVE Rb,x(Ra)
```

Основная идея подхода заключается в том, что каждая команда машины описывается в виде такого образца. Различные покрытия дерева промежуточного представления соответствуют различным последовательностям машинных команд. Задача выбора команд состоит в том, чтобы выбрать наилучший способ реализации того или иного действия или последовательности действий, т. е. выбрать в некотором смысле оптимальное покрытие.

Для выбора оптимального покрытия было предложено несколько интересных алгоритмов, в частности использующих динамическое программирование [11, 13]. Мы здесь рассмотрим алгоритм [12], комбинирующий возможности синтаксического анализа и динамического программирования. В основу этого алгоритма положен синтаксический анализ неоднозначных грамматик (модифицированный алгоритм Кока, Янгера и Касами [15, 16]), эффективный в реальных приложениях. Этот же метод может быть применен и тогда, когда в качестве промежуточного представления используется дерево.

9.9.2 Синтаксический анализ для Т-грамматик

Обычно код генерируется из некоторого промежуточного языка с довольно жесткой структурой. В частности, для каждой операции известна ее размерность, то есть число операндов, большее или равное 0. Операции задаются терминальными символами, и наоборот – будем считать все терминальные символы знаками операций. Назовем грамматики, удовлетворяющие этим ограничениям, *Т-грамматиками*. Правая часть каждой продукции в Т-грамматике есть правильное префиксное выражение, которое может быть задано следующим определением:

(1) Операция размерности 0 является правильным префиксным выражением;

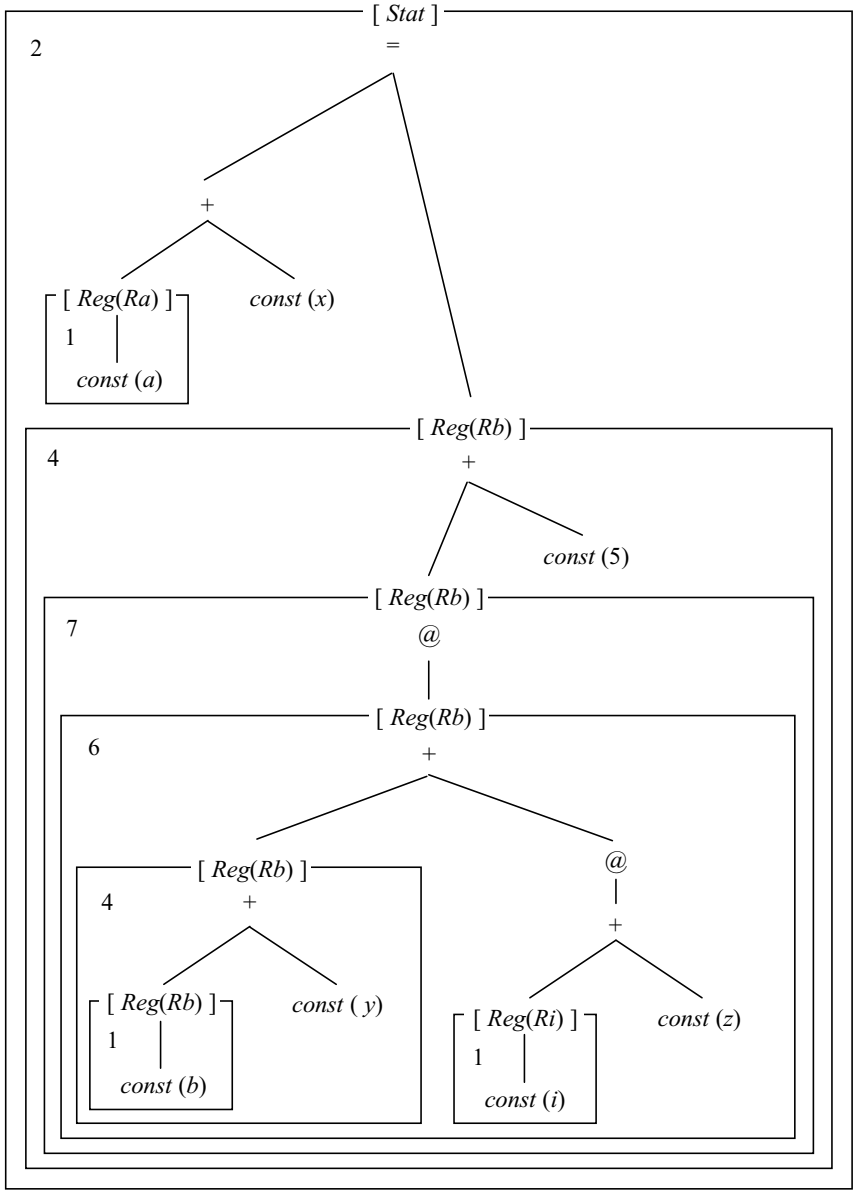


Рис. 9.17:

(2) Нетерминал является правильным префиксным выражением;

(3) Префиксное выражение, начинающееся со знака операции размерности $n > 0$, является правильным, если после знака операции сле-

дует n правильных префиксных выражений;

(4) Ничто другое не является правильным префиксным выражением.

Образцы, соответствующие машинным командам, задаются правилами грамматики (вообще говоря, неоднозначной). Генератор кода анализирует входное префиксное выражение и строит одновременно все возможные деревья разбора. После окончания разбора выбирается дерево с наименьшей стоимостью. Затем по этому единственному оптимальному дереву генерируется код.

Для Т-грамматик все цепочки, выводимые из любого нетерминала A, являются префиксными выражениями с фиксированной арностью операций. Длины всех выражений из входной цепочки $a_1 \ldots a_n$ можно предварительно вычислить (под длиной выражения имеется ввиду длина подстроки, начинающейся с символа кода операции и заканчивающейся последним символом, входящим в выражение для этой операции). Поэтому можно проверить, сопоставимо ли некоторое правило с подцепочкой $a_i \ldots a_k$ входной цепочки $a_1 \ldots a_n$, проходя слева-направо по $a_i \ldots a_k$. В процессе прохода по цепочке предварительно вычисленные длины префиксных выражений используются для того, чтобы перейти от одного терминала к следующему терминалу, пропуская подцепочки, соответствующие нетерминалам правой части правила.

Цепные правила не зависят от операций, следовательно, их необходимо проверять отдельно. Применение одного цепного правила может зависеть от применения другого цепного правила. Следовательно, применение цепных правил необходимо проверять до тех пор, пока нельзя применить ни одно из цепных правил. Мы предполагаем, что в грамматике нет циклов в применении цепных правил. Построение всех вариантов анализа для Т-грамматики дано ниже в алгоритме 9.1. Тип Titem в алгоритме 9.1 ниже служит для описания ситуаций (т.е. правил вывода и позиции внутри правила). Тип Tterminal – это тип терминального символа грамматики, тип Tproduction – тип для правила вывода.

Алгоритм 9.1

```
Tterminal a[n];
setofTproduction r[n];
int l[n];  // l[i] - длина a[i]-выражения
Titem h;   // используется при поиске правил,
        // сопоставимых с текущей подцепочкой
// Предварительные вычисления
Для каждой позиции i вычислить длину a[i]-выражения l[i];
// Распознавание входной цепочки
  for (int i=n-1;i>=0;i--){
    for (каждого правила A -> a[i] y из P){
      j=i+1;
      if (l[i]>1)
```

```
    // Первый терминал a[i] уже успешно сопоставлен
      {h=[A->a[i].y];
       do // найти a[i]y, сопоставимое с a[i]..a[i+l[i]-1]
        {Пусть h==[A->u.Xv];
         if (X in T)
          if (X==a[j]) j=j+1; else break;
         else // X in N
          if (имеется X->w in r[j]) j=j+l[j]; else break;
         h=[A->uX.v];
         //перейти к следующему символу
      }
        while (j==i+l[i]);
       } // l[i]>1
     if (j==i+l[i]) r[i]=r[i] + { (A->a[i]y) }
    } // for по правилам A -> a[i] y
    // Сопоставить цепные правила
    while (существует правило C->A из P такое, что
        имеется некоторый элемент (A->w) в r[i]
        и нет элемента (C->A) в r[i])
     r[i]=r[i] + { (C->A) };
  } // for по i
  Проверить, принадлежит ли (S->w) множеству r[0];
```

Множества $r[i]$ имеют размер $O(|P|)$. Можно показать, что алгоритм имеет временную и емкостную сложность $O(n)$.

Рассмотрим вновь пример рис. 9.15. В префиксной записи приведенный фрагмент программы записывается следующим образом:

```
= + a x + @ + + b y @ + i z 5
```

На рис. 9.18 приведен результат работы алгоритма. Правила вычисления стоимости приведены в следующем разделе. Все возможные выводы входной цепочки (включая оптимальный) можно построить, используя таблицу l длин префиксных выражений и таблицу r применимых правил.

Пусть G – это T-грамматика. Для каждой цепочки z из $L(G)$ можно построить абстрактное синтаксическое дерево соответствующего выражения (рис. 9.15). Мы можем переписать алгоритм так, чтобы он принимал на входе абстрактное синтаксическое дерево выражения, а не цепочку. Этот вариант алгоритма приведен ниже. В этом алгоритме дерево выражения обходится сверху вниз и в нем ищутся поддеревья, сопоставимые с правыми частями правил из G. Обход дерева осуществляется процедурой PARSE. После обхода поддерева данной вершины в ней применяется процедура MATCHED, которая пытается найти все образцы, сопоставимые поддереву данной вершины. Для этого каждое правило-образец разбивается на компоненты в соответствии с встречающимися в нем операциями. Дерево обходится справа налево только для того, что-

Операция	Длина	Правила (стоимость)	
=	15	2(22)	
+	3	4(5)	5(6)
a	1	1(2)	
x	1	1(2)	
+	11	4(14)	5(17)
@	9	7(11)	
+	8	5(11)	6(9)
+	3	4(5)	5(6)
b	1	1(2)	
y	1	1(2)	
@	4	7(7)	3(4)
+	3	4(5)	5(6)
i	1	1(2)	
z	1	1(2)	
5	1	1(2)	

Рис. 9.18:

бы иметь соответствие с порядком вычисления в алгоритме 9.1. Очевидно, что можно обходить дерево вывода и слева направо.

Структура данных, представляющая вершину дерева, имеет следующую форму:

```
struct Tnode {
  Tterminal op;
  Tnode * son[MaxArity];
  setofTproduction RULEs;
};
```

В комментариях указаны соответствующие фрагменты алгоритма 9.1.

Алгоритм 9.2

```
Tnode * root;

bool MATCHED(Tnode * n, Titem h)
{ bool matching;
  пусть h==[A->u.Xvy], v== v1 v2 ... vm, m=Arity(X);
  if (X in T)// сопоставление правила
    if (m==0)  // if l[i]==1
      if (X==n->op)   //if X==a[j]
        return(true);
      else
        return(false);
```

```
    else // if l[i]>1
     if (X==n->op) //if X==a[j]
       {matching=true;
        for (i=1;i<=m;i++) //j=j+l[j]
        matching=matching &&   //while (j==i+l[i])
              MATCHED(n->son[i-1],[A->uXv'.vi v"y]);
        return(matching);  //h=[A->uX.v]
          }
     else
       return(false);
   else // X in N поиск подвывода
     if (в n^.RULEs имеется правило с левой частью X)
       return(true);
     else
       return(false);
 }

void PARSE(Tnode * n)
 {
  for (i=Arity(n->op);i>=1;i--)
  // for (i=n; i>=1;i--)
     PARSE(n->son[i-1]);
  n->RULEs=EMPTY;
  for (каждого правила A->bu из P такого, что b==n->op)
     if (MATCHED(n,[A->.bu]))   //if (j==i+l[i])
      n->RULEs=n->RULEs+{(A->bu)};
 // Сопоставление цепных правил
  while (существует правило C->A из P такое, что
        некоторый элемент (A->w) в n->RULEs
        и нет  элемента   (C->A) в n->RULEs)
    n->RULEs=n->RULEs+{(C->A)};
 }

   Основная программа

 // Предварительные вычисления
   Построить дерево выражения для входной цепочки z;
   root = указатель дерева выражения;
 // Распознать входную цепочку
   PARSE(root);
   Проверить, входит ли во множество root->RULEs
   правило с левой частью S;
```

Выходом алгоритма является дерево выражения для z, вершинам которого сопоставлены применимые правила. С помощью такого дерева можно построить все выводы для исходного префиксного выражения.

9.9.3 Выбор дерева вывода наименьшей стоимости

Т-грамматики, описывающие системы команд, обычно являются неоднозначными. Чтобы сгенерировать код для некоторой входной цепочки, необходимо выбрать одно из возможных деревьев вывода. Это дерево должно представлять желаемое качество кода, например размер кода и/или время выполнения.

Для выбора дерева из множества всех построенных деревьев вывода можно использовать атрибуты стоимости, атрибутные формулы, вычисляющие их значения, и критерии стоимости, которые оставляют для каждого нетерминала единственное применимое правило. Атрибуты стоимости сопоставляются всем нетерминалам, атрибутные формулы – всем правилам Т-грамматики.

Предположим, что для вершины n обнаружено применимое правило

$$p : A \to z_0 X_1 z_1 \ldots X_k z_k,$$

где $z_i \in T^*$ для $0 \leqslant i \leqslant k$ и $X_j \in N$ для $1 \leqslant j \leqslant k$. Вершина n имеет потомков $n_1, \ldots, n_k$, которые соответствуют нетерминалам $X_1, \ldots, X_k$. Значения атрибутов стоимости вычисляются обходя дерево снизу вверх. Вначале атрибуты стоимости инициализируются неопределенным значением UndefinedValue. Предположим, что значения атрибутов стоимости для всех потомков $n_1, \ldots, n_k$ вершины n вычислены. Если правилу p сопоставлена формула

$$a(A) = f(b(X_i), c(X_j), \ldots) \text{ для } 1 \leqslant i, j \leqslant k,$$

то производится вычисление значения атрибута a нетерминала A в вершине n. Для всех примененных правил ищется такое, которое дает минимальное значение стоимости. Отсутствие примененных правил обозначается через Undefined, значение которого полагается большим любого определенного значения.

Добавим в алгоритм 9.2 реализацию атрибутов стоимости, формул их вычисления и критериев отбора. Из алгоритма можно исключить поиск подвыводов, соответствующих правилам, для которых значение атрибута стоимости не определено. Структура данных, представляющая вершину дерева, принимает следующий вид:

```
struct Tnode {
  Tterminal op;
  Tnode * son[MaxArity];
  struct * { unsigned CostAttr;
          Tproduction Production;
        } nonterm [Tnonterminal];
OperatorAttributes ...
```

Тело процедуры PARSE принимает вид

```
{for (i=Arity(n->op);i>=1;i--)
   PARSE(n->son[i]);
   for (каждого $A$ из N)
    {n->nonterm[A].CostAttr=UndefinedValue;
     n->nonterm[A].production=Undefined;
    }
   for (каждого правила $A->bu$ из P
       такого, что b==n->op)
    if (MATCHED(n,[A->.bu]))
     {ВычислитьАтрибутыСтоимостиДля(A,n,(A->bu));
      ПроверитьКритерийДля(A,n->nonterm[A].CostAttr);
       if ((A->bu) лучше,
          чем ранее обработанное правило для A)
         {Модифицировать(n->nonterm[A].CostAttr);
          n->nonterm[A].production=(A->bu);
         }
     }
// Сопоставить цепные правила
while (существует правило $C->A$ из $P$, которое
      лучше, чем ранее обработанное правило для A)
 {ВычислитьАтрибутыСтоимостиДля(C,n,(C->A));
  ПроверитьКритерийДля(C,n->nonterm[C].CostAttr);
  if ((C->A) лучше)
   {Модифицировать(n->nonterm[C].CostAttr);
    n->nonterm[C].production=(C->A);
   }
 }
}
```

Дерево наименьшей стоимости определяется как дерево, соответствующее минимальной стоимости корня. Когда выбрано дерево вывода наименьшей стоимости, вычисляются значения атрибутов, сопоставленных вершинам дерева вывода, и генерируются соответствующие машинные команды. Вычисление значений атрибутов, генерация кода осуществляются в процессе обхода выбранного дерева вывода сверху вниз, слева направо. Обход выбранного дерева вывода выполняется процедурой вычислителя атрибутов, на вход которой поступают корень дерева выражения и аксиома грамматики. Процедура использует правило $A \to z_0 X_1 z_1 \ldots X_k z_k$, связанное с указанной вершиной n, и заданный нетерминал A, чтобы определить соответствующие им вершины $n_1, \ldots, n_k$ и нетерминалы $X_1, \ldots, X_k$. Затем вычислитель рекурсивно обходит каждую вершину n_i, имея на входе нетерминал X_i.

9.9.4 Атрибутная схема для алгоритма сопоставления образцов

Алгоритмы 9.1 и 9.2 являются "универсальными" в том смысле, что конкретные грамматики выражений и образцов являются, по-существу, параметрами этих алгоритмов. В то же время, для каждой конкретной грамматики можно написать свой алгоритм поиска образцов. Например, в случае нашей грамматики выражений и приведенных на рис. 9.16 образцов алгоритм 9.2 может быть представлен атрибутной грамматикой, приведенной ниже.

Наследуемый атрибут Match содержит упорядоченный список (вектор) образцов для сопоставления в поддереве данной вершины. Каждый из образцов имеет вид либо $<op\ op{-}list>$ (op – операция в данной вершине, а $op{-}list$ – список ее операндов), либо представляет собой нетерминал N. В первом случае $op{-}list$ "распределяется" по потомкам вершины для дальнейшего сопоставления. Во втором случае сопоставление считается успешным, если есть правило $N \rightarrow op\ \{Pat_i\}$, где w состоит из образцов, успешно сопоставленных потомкам данной вершины. В этом случае по потомкам в качестве образцов распределяются элементы правой части правила. Эти два множества образцов могут пересекаться. Синтезируемый атрибут Pattern – вектор логических значений, дает результат сопоставления по вектору-образцу Match.

Таким образом, при сопоставлении образцов могут встретиться два случая:

1. Вектор образцов содержит образец $<op\ \{Pat_i\}>$, где op – операция, примененная в данной вершине. Тогда распределяем образцы Pat_i по потомкам и сопоставление по данному образцу считаем успешным (истинным), если успешны сопоставления элементов этого образца по всем потомкам.

2. Образцом является нетерминал N. Тогда рассматриваем все правила вида $N \rightarrow op\ \{Pat_i\}$. Вновь распределяем образцы Pat_i по потомкам и сопоставление считаем успешным (истинным), если успешны сопоставления по всем потомкам. В общем случае успешным может быть сопоставление по нескольким образцам.

Отметим, что в общем случае в потомки одновременно передается несколько образцов для сопоставления.

В приведенной ниже атрибутной схеме не рассматриваются правила выбора покрытия наименьшей стоимости (см. предыдущий раздел). Выбор оптимального покрытия может быть сделан еще одним проходом по дереву, аналогично тому, как это было сделано выше. Например, в правиле с '+' имеется несколько образцов для Reg, но реального выбора одного из них не осуществляется. Кроме того, не уточнены некоторые детали реализации. В частности, конкретный способ формирования векторов Match и Pattern. В тексте употребляется термин "добавить", что означает

добавление к вектору образцов очередного элемента. Векторы образцов записаны в угловых скобках.

```
RULE
Stat ::= '=' Reg Reg

SEMANTICS
Match<2>=<'+' Reg Const>;
Match<3>=<Reg>;
Pattern<0>[1]=Pattern<2>[1]&Pattern<3>[1].
```

Этому правилу соответствует один образец 2. Поэтому в качестве образцов потомков через их атрибуты Match передаются, соответственно, <'+' Reg Const> и <Reg>.

```
RULE
Reg ::= '+' Reg Reg

SEMANTICS
if (Match<0> содержит Reg в позиции i)
   {Match<2>=<Reg,Reg,Reg>;
    Match<3>=<Const,Reg,<'@' '+' Reg Const>>;
   }
if (Match<0> содержит образец <'+' Reg Const> в позиции j)
   {добавить Reg к Match<2> в некоторой позиции k;
   добавить Const к Match<3> в некоторой позиции k;
   }
if (Match<0> содержит образец <'+' Reg Const> в позиции j)
Pattern<0>[j]=Pattern<2>[k]&Pattern<3>[k];
if (Match[0] содержит Reg в i-й позиции)
Pattern<0>[i]=(Pattern<2>[1]&Pattern<3>[1])
         |(Pattern<2>[2]&Pattern<3>[2])
         |(Pattern<2>[3]&Pattern<3>[3]).
```

Образцы, соответствующие этому правилу, следующие:

(4) $Reg \rightarrow$ '+' $Reg\ Const$,

(5) $Reg \rightarrow$ '+' $Reg\ Reg$,

(6) $Reg \rightarrow$ '+' Reg '@' '+' $Reg\ Const$.

Атрибутам Match второго и третьего символов в качестве образцов при сопоставлении могут быть переданы векторы <Reg, Reg, Reg> и <Const, Reg, <'@' '+' Reg Const>>, соответственно. Из анализа других правил можно заключить, что при сопоставлении образцов предков левой части данного правила атрибуту Match символа левой части может быть передан образец <'+' Reg Const> (из образцов 2, 3, 6) или образец Reg.

```
RULE
Reg ::= '@' Reg
```

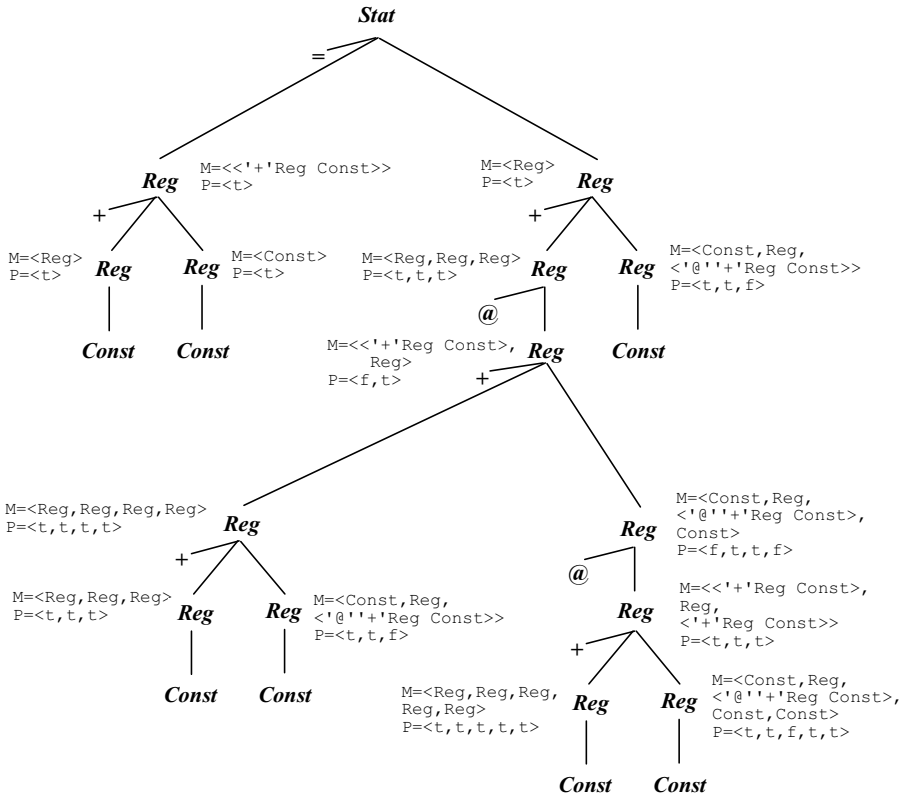


Рис. 9.19:

SEMANTICS
if (Match<0> содержит Reg в i-й позиции)
  Match<2>=<'+' Reg Const>,Reg>;
if (Match<0> содержит <'@' '+' Reg Const> в j-й позиции)
  добавить к Match<2> <'+' Reg Const> в k позиции;
if (Match<0> содержит Reg в i-й позиции)
  Pattern<0>[i]=Pattern<2>[1]|Pattern<2>[2];
if (Match<0> содержит <'@' '+' Reg Const> в j-й позиции)
  Pattern<0>[j]=Pattern<2>[k].

Образцы, соответствующие этому правилу, следующие:
  (3) $Reg \rightarrow$ '@' '+' $Reg\ Const$,
  (7) $Reg \rightarrow$ '@' Reg.
Соответственно, атрибуту Match второго символа в качестве образцов при сопоставлении могут быть переданы <'+' Reg Const> (образец 3) или <Reg> (образец 7). Из анализа других правил можно заключить, что при со-

поставлении образцов предков левой части данного правила атрибуту Match могут быть переданы образцы <'@' '+' Reg Const> (из образца 6) и Reg.

RULE
Reg ::= Const

SEMANTICS
if (Pattern<0> содержит Const в j-й позиции)
Pattern<0>[j]=true;
if (Pattern<0> содержит Reg в i-й позиции)
Pattern<0>[i]=true.

Для дерева рис. 9.15 получим значения атрибутов, приведенные на рис. 9.19. Здесь M обозначает Match, P – Pattern, C – Const, R – Reg.

Глава 10

Системы автоматизации построения трансляторов

Системы автоматизации построения трансляторов (САПТ) предназначены для автоматизации процесса разработки трансляторов. Очевидно, что для того, чтобы описать транслятор, необходимо иметь формализм для описания. Этот формализм затем реализуется в виде входного языка САПТ. Как правило, формализмы основаны на атрибутных грамматиках. Ниже описаны две САПТ, получившие распространение: СУПЕР [3] и Yacc. В основу первой системы положены LL(1)-грамматики и L-атрибутные вычислители, в основу второй – LALR(1)-грамматики и S-атрибутные вычислители.

10.1 Система СУПЕР

Программа на входном языке СУПЕР ("метапрограмма") состоит из следующих разделов:

- Заголовок;
- Раздел констант;
- Раздел типов;
- Алфавит;
- Раздел файлов;
- Раздел библиотеки;
- Атрибутная схема.

Заголовок определяет имя атрибутной грамматики, первые три буквы имени задают расширение имени входного файла для реализуемого транслятора.

Раздел констант содержит описание констант, раздел типов – описание типов.

Алфавит содержит перечисление нетерминальных символов и классов лексем, а также атрибутов (и их типов), сопоставленных этим симво-

лам. Классы лексем являются терминальными символами с точки зрения синтаксического анализа, но могут иметь атрибуты, вычисляемые в процессе лексического анализа. Определение класса лексем состоит в задании имени класса, имен атрибутов для этого класса и типов этих атрибутов.

В разделе определения нетерминальных символов содержится перечисление этих символов с указанием приписанных им атрибутов и их типов. Аксиома грамматики указывается первым символом в списке нетерминалов.

Раздел библиотеки содержит заголовки процедур и функций, используемых в формулах атрибутной грамматики.

Раздел файлов содержит описание файловых переменных, используемых в формулах атрибутной грамматики. Файловые переменные можно рассматривать как атрибуты аксиомы.

Атрибутная схема состоит из списка синтаксических правил и сопоставленных им семантических правил. Для описания синтаксиса языка используется расширенная форма Бэкуса-Наура. Терминальные символы в правой части заключаются в кавычки, классы лексем и нетерминалы задаются их именами. Для задания в правой части необязательных символов используются скобки [ ], для задания повторяющихся конструкций используются скобки ( ). В этом случае может быть указан разделитель символов (после /). Например,

```
A ::= B [ C ] ( D ) ( E / ',' )
```

Первым правилом в атрибутной схеме должно быть правило для аксиомы.

Каждому синтаксическому правилу могут быть сопоставлены семантические действия. Каждое такое действие – это оператор, который может использовать атрибуты как символов данного правила (локальные атрибуты), так и символов, могущих быть предками (динамически) символа левой части данного правила в дереве разбора (глобальные атрибуты). Для ссылки на локальные атрибуты символы данного правила (как терминальные, так и нетерминальные) нумеруются от 0 (для символа левой части). При ссылке на глобальные атрибуты надо иметь в виду, что атрибуты имеют области видимости на дереве разбора. Областью видимости атрибута вершины, помеченной N, является все поддерево N, за исключением его поддеревьев, также помеченных N.

Исполнение операторов семантической части правила привязывается к обходу дерева разбора сверху вниз слева направо. Для этого каждый оператор может быть помечен меткой, состоящей из номера ветви правила, к выполнению которой должен быть привязан оператор, и, возможно, одного из суффиксов A, B, E, M.

Суффикс A задает выполнение оператора перед каждым вхождением синтаксической конструкции, заключенной в скобки повторений ( ). Суффикс B задает выполнение оператора после каждого вхождения синтаксической конструкции, заключенной в скобки повторений ( ). Суф-

фикс M задает выполнение оператора между вхождениями синтаксической конструкции, заключенной в скобки повторений ( ). Суффикс E задает выполнение оператора в том случае, когда конструкция, заключенная в скобки [ ], отсутствует.

Пример использование меток атрибутных формул:

```
D ::= 'd' =>
    $0.y:=$0.x+1.
A ::= B (C) [D] =>
    $2.x:=1;
 2M: $2.x:=$2.x+1;
    $3.x:=$2.x;
 3E: $3.y:=$3.x;
 3:  writeln($3.y).
```

Процедура writeln напечатает число вхождений символа C в C-список, если D опущено. В противном случае напечатанное число будет на единицу больше.

10.2 Система Yacc

В основу системы Yacc положен синтаксический анализатор типа LALR(1), генерируемый по входной (мета) программе. Эта программа состоит из трех частей:

```
%{
Си-текст
%}
%token Список имен лексем
%%
Список правил трансляции
%%
Служебные Си-подпрограммы
```

Си-текст (который вместе с окружающими скобками %{ и %} может отсутствовать) обычно содержит Си-объявления (включая #include и #define), используемые в тексте ниже. Этот Си-текст может содержать и объявления (или предобъявления) функций.

Список имен лексем содержит имена, которые преобразуются Yacc-препроцессором в объявления констант (#define). Как правило, эти имена используются как имена классов лексем и служат для определения интерфейса с лексическим анализатором.

Каждое правило трансляции имеет вид

```
Левая_часть : альтернатива_1 {семантические_действия_1}
 | альтернатива_2 {семантические_действия_2}
```

```
|...
| альтернатива_n {семантические_действия_n}
;
```

Каждое семантическое действие – это последовательность операторов Си. При этом каждому нетерминалу может быть сопоставлен один синтезируемый атрибут. На атрибут нетерминала левой части ссылка осуществляется посредством значка $$, на атрибуты символов правой части – посредством значков $1, $2 , . . . , $n, причем номер соответствует порядку элементов правой части, включая семантические действия. Каждое семантическое действие может вырабатывать значение в результате выполнения присваивания $$=Выражение. Исполнение текого оператора в последнем семантическом действии определяет значение атрибута символа левой части.

В некоторых случаях допускается использование грамматик, имеющих конфликты. При этом синтаксический анализатор разрешает конфликты следующим образом:

– конфликты типа свертка/свертка разрешаются выбором правила, предшествующего во входной метапрограмме;

– конфликты типа сдвиг/свертка разрешаются предпочтением сдвига. Поскольку этих правил не всегда достаточно для правильного определения анализатора, допускается определение старшинства и ассоциативности терминалов.

Например, объявление

```
%left '+' '-'
```

определяет $+$ и $-$, имеющими одинаковый приоритет и имеющими левую ассоциативность. Операцию можно определить как правоассоциативную в результате объявления:

```
%right '^'
```

Бинарную операцию можно определить как неассоциативную (т.е. не допускающую появления объединения двух подряд идущих знаков операции):

```
%nonassoc '<'
```

Символы, перечисленные в одном объявлении, имеют одинаковое старшинство. Старшинство выше для каждого последующего объявления. Конфликты разрешаются путем присваивания старшинства и ассоциативности каждому правилу грамматики и каждому терминалу, участвующим в конфликте. Если необходимо выбрать между сдвигом входного символа s и сверткой по правилу $A \to w$, свертка делается, если старшинство правила больше старшинства s или если старшинство одинаково, а правило левоассоциативно. В противном случае делается сдвиг.

Обычно за старшинство правила принимается старшинство самого правого терминала правила. В тех случаях, когда самый правый терминал не дает нужного приоритета, этот приоритет можно назначить следующим объявлением:

```
%prec терминал
```

Старшинство и ассоциативность правила в этом случае будут такими же, как у указанного терминала.

Yacc не сообщает о конфликтах, разрешаемых с помощью ассоциативности и приоритетов. Восстановление после ошибок управляется пользователем с помощью введения в грамматику "правил ошибки" вида

$A \to$ error w.

Здесь error – ключевое слово Yacc. Когда встречается синтаксическая ошибка, анализатор трактует состояние, набор ситуаций для которого содержит правило для error, некоторым специальным образом: символы из стека выталкиваются до тех пор, пока на верхушке стека не будет обнаружено состояние, для которого набор ситуаций содержит ситуацию вида $[A \to .\text{error } w]$. После чего в стек фиктивно помещается символ error, как если бы он встретился во входной строке.

Если w пусто, делается свертка. После этого анализатор пропускает входные символы, пока не найдет такой, с которым можно продолжить нормальный разбор.

Если w не пусто, просматривается входная строка и делается попытка свернуть w. Если w состоит только из терминалов, эта строка ищется во входном потоке.

Литература

[1] Адельсон-Вельский Г.М., Ландис Е.М. *Один алгоритм организации информации*. ДАН СССР. 1962. Т.146. N 2. С. 263-266.

[2] Ахо А., Ульман Д. *Теория синтаксического анализа, перевода и компиляции, в 2-х т.* М.: Мир, 1978.

[3] Бездушный А.Н., Лютый В.Г., Серебряков В.А. *Разработка компиляторов в системе* СУПЕР. М.: ВЦ АН СССР, 1991.

[4] Грис Д. *Конструирование компиляторов для цифровых вычислительных машин*. М.: Мир, 1975.

[5] Кнут Д. *Искусство программирования для* ЭВМ. *Т.1. Основные алгоритмы*. М.: Мир, 1976.

[6] Кнут Д. *Семантика контекстно-свободных языков*. В сб.: Семантика языков программирования. М.: Мир, 1980.

[7] Курочкин В.М. *Алгоритм распределения регистров для выражений за один обход дерева вывода*. 2 Всес. конф "Автоматизация производства ППП и трансляторов". 1983. С. 104-105.

[8] Лавров С.С., Гончарова Л.И. *Автоматическая обработка данных. Хранение информации в памяти* ЭВМ. М.: Наука, 1971.

[9] Надежин Д.Ю., Серебряков В.А., Ходукин В.М. *Промежуточный язык Лидер (предварительное сообщение)*. Обработка символьной информации. М.: ВЦ АН СССР, 1987. С. 50-63.

[10] Aho A., Sethi R., Ullman J. *Compilers: principles, techniques and tools*. N.Y.: Addison-Wesley, 1986.

[11] Aho A.U., Ganapathi M., Tjiang S.W. *Code generation using tree matching and dynamic programing*. ACM Trans. Program. Languages and Systems. 1989. V.11.N 4.

[12] Bezdushny A., Serebriakov V. *The use of the parsing method for optimal code generation and common subexpression elimination*. Techn. et Sci. Inform. 1993. V.12. N.1. P.69-92.

[13] Emmelman H., Schroer F.W., Landweher R. BEG – *a generator for efficient back-ends*. ACM SIGPLAN. 1989. V.11. N 4.p.227-237

[14] Fraser C.W., Hanson D.R. A *Retargetable compiler for ANSI* C. SIGPLAN Notices. 1991. V 26.

[15] Graham S.L., Harrison M.A., Ruzzo W.L. *An improved context-free recognizer*. ACM Trans. Program. Languages and Systems. 1980. N.2.

[16] Harrison M.A. *Introduction to formal language theory*. Reading, Mass.: Addison-Wesley, 1978.