В.А.Серебряков

Лекции по конструированию компиляторов

Москва
1997

Предисловие

Предлагаемая вниманию читателя книга основана на курсе лекций, прочитанных автором на факультете вычислительной математики и кибернетики Московского государственного университета в 1991-1993 гг. Автор надеется, что издание книги восполнит существенный пробел в литературе на руссом языке по разработке компиляторов.

Содержание книги представляет собой "классические" разделы предмета: лексический и синтаксический анализ, организация памяти транслятора (таблицы символов) и периода исполнения (магазина), генерация кода, в частности генерация арифметических и логических выражений. 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. Автор надеется восполнить эти пробелы в будущем.

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

Глава 1. Введение 6
1.1. Место компилятора в программном обеспечении 6
1.2. Структура компилятора 7
Глава 2. Лексический анализ 11
2.1. Регулярные множества и регулярные выражения 13
2.2. Конечные автоматы 14
2.3. Построение детерминированного конечного автомата по недетерминированному 16
2.4. Построение детерминированного конечного автомата по регулярному выражению 18
2.5. Построение детерминированного конечного автомата с минимальным числом состояний 21
2.6. Программирование лексических анализаторов 23
2.7. Конструктор лексических анализаторов LEX 27
Глава 3. Синтаксический анализ 32
3.1. Основные понятия и определения 32
3.2. Таблично-управляемый предсказывающий разбор 34
3.2.1. Алгоритм разбора сверху-вниз 34
3.2.2. Множества FIRST и FOLLOW 38
3.2.3. Конструирование таблиц предсказывающего анализатора 40
3.2.4. LL(1)-грамматики 41
3.2.5. Удаление левой рекурсии 42
3.2.6. Левая факторизация 44
3.2.7. Рекурсивный спуск 45
3.2.8. Диаграммы переходов для рекурсивного спуска 46
3.2.9. Восстановление после синтаксических ошибок 49
3.3. Разбор снизу-вверх типа сдвиг-свертка 50
3.3.1. Основа 50
3.3.2. LR(k)-анализаторы 52
3.3.3. LR грамматики 57
3.3.4. Конфликты разбора типа сдвиг-свертка 63
3.3.5. Восстановление после синтаксических ошибок 64
Глава 4. Элементы теории перевода 65
4.1. Преобразователи с магазинной памятью 65
4.2. Синтаксически управляемый перевод 66
4.3. Атрибутные грамматики 70
4.3.1. Определение атрибутных грамматик 70
4.3.2. Атрибутированное дерево разбора 71
4.3.3. Язык описания атрибутных грамматик 72
4.3.4. Классы атрибутных грамматик и их реализация 75
Глава 5. Контекстные условия языков программирования 77
5.1. Описание областей видимости и блочной структуры 77
5.2. Структура среды Модулы-2 78
5.3. Занесение в среду и поиск объектов 81

Глава 6. Организация таблиц символов компилятора 89
6.1. Таблицы идентификаторов и таблицы символов 89
6.2. Таблицы идентификаторов 90
6.3. Таблицы символов и таблицы расстановки 93
6.4. Функции расстановки 94
6.5. Таблицы на деревьях 96
6.6. Реализация блочной структуры 100
6.7. Сравнение различных методов реализации таблиц 100
Глава 7. Промежуточные представления программы 102
7.1. Представление в виде ориентированного графа 102
7.2. Трехадресный код 102
7.3. Линеаризованные представления 107
7.4. Виртуальная машина Java 109
7.5. Организация информации в генераторе кода 113
7.6. Уровень промежуточного представления 115
Глава 8. Генерация кода 116
8.1. Модель машины 116
8.2. Динамическая организация памяти 119
8.2.1. Организация магазина со статической цепочкой 120
8.2.1. Организация магазина с дисплеем 124
8.3. Назначение адресов 126
8.4. Трансляция переменных 128
8.5. Трансляция целых выражений 134
8.6. Распределение регистров при вычислении арифметических выражений 136
8.7. Трансляция логических выражений 145
8.8. Выделение общих подвыражений 153
8.9. Генерация оптимального кода методами синтаксического анализа 157
8.9.1. Сопоставление образцов 157
8.9.2. Синтаксический анализ для Т-грамматик 160
8.9.3. Выбор дерева вывода наименьшей стоимости 168
Литература 171

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

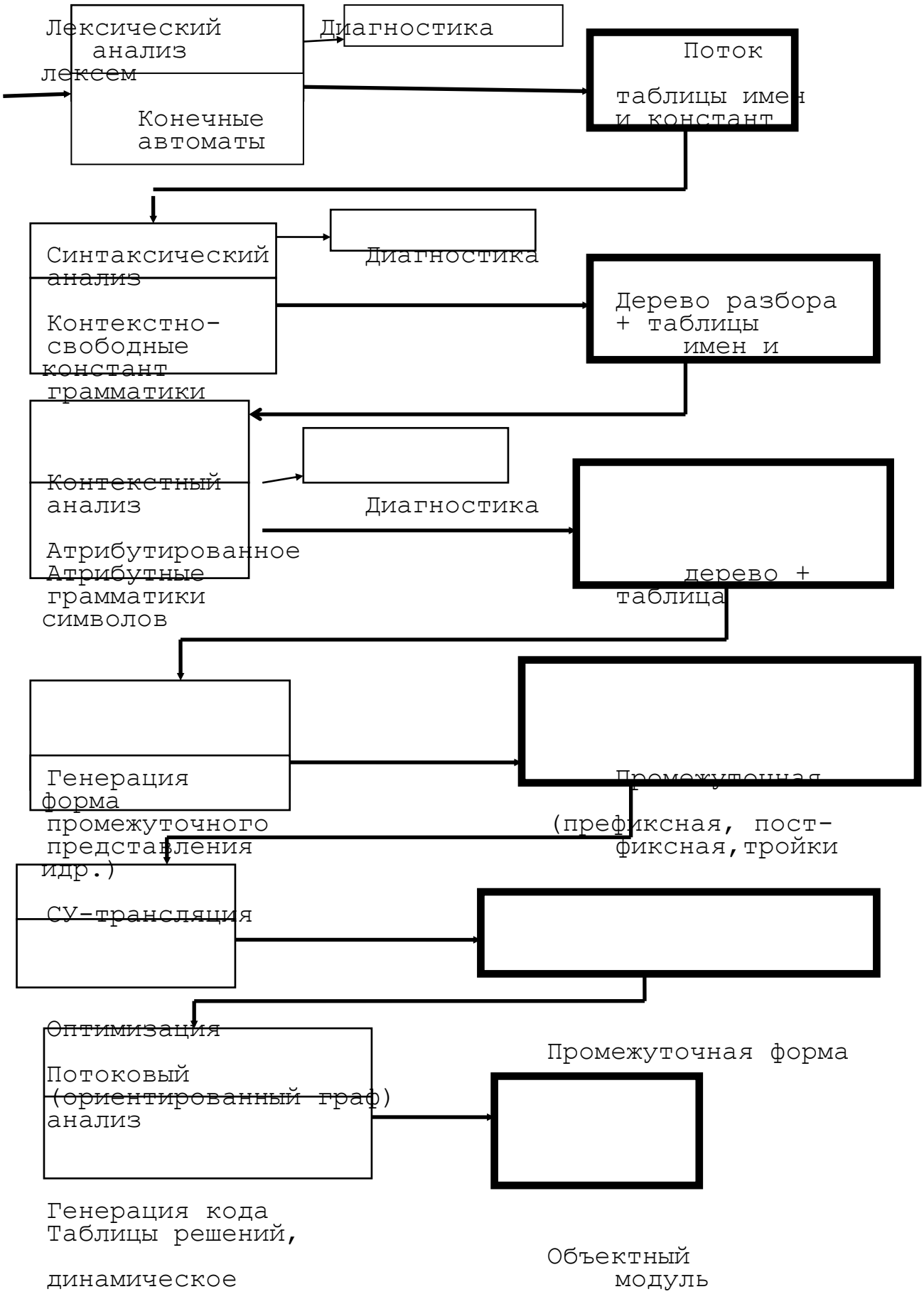
Наконец, бурно развиваются различные параллельные архитектуры. Среди них отметим векторные, многопроцессорные, с широким командным словом (вариантом которых являются суперскалярные ЭВМ). На рынке уже имеются десятки типов ЭВМ с параллельной архитектурой, начиная от супер-ЭВМ (Cray, CDC и другие), через 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 (ЛА) входная программа, представляющая собой поток символов, разбивается на лексемы - слова в соответствии с определениями языка. Основным формализмом, 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 за очередной лексемой, либо как полный проход, результатом которого является файл лексем.

В процессе выделения лексем ЛА может как самостоятельно строить таблицы имен и констант, так и выдавать значения для каждой лексемы при очередном обращении к нему. В этом случае таблица имен строится в последующих фазах (например, в процессе синтаксического анализа).


Лексический анализ лексем
Конечные автоматы
Диагностика
Поток
таблицы имен и констант
Синтаксический анализ
Контекстно-свободные грамматики
Диагностика
Дерево разбора + таблицы имен и констант
Контекстный анализ
Атрибутированное Атрибутные грамматики символов
Диагностика
дерево + таблица
Генерация форма промежуточного представления идр.)
Промежуточная (префиксная, пост-фиксная,тройки
СУ-трансляция
Оптимизация
Потоковый (ориентированный граф) анализ
Промежуточная форма
Генерация кода Таблицы решений,
динамическое
Объектный модуль

программирование и др

Рис. 1.1

На этапе ЛА 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 использовании систем автоматизации построения синтаксических анализаторов.

Результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицу имен. 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 строится таблица символов, которую можно рассматривать как таблицу имен, пополненную информацией об описаниях (свойствах) объектов.

Основным формализмом, использующимся при контекстном анализе, являются атрибутные грамматики. Результатом работы фазы контекстного анализа является атрибутированное дерево программы. Информация об объектах может быть как рассредоточена в самом дереве, так и сосредоточена в отдельных таблицах символов. 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, различные 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.

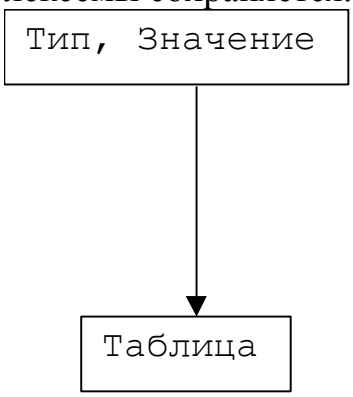
Глава 2. Лексический анализ

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 (пробел 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 контексного анализа, работающего вслед за синтаксическим, важна информация о конкретных значениях отдельных лексем (идентификаторов, чисел и т.д.). Поэтому общая схема работы лексического анализатора такова. Сначала выделяем отдельную лексему (возможно, используя символы-разделители). Если выделенная лексема - ограничитель, то он (точнее, некоторый его признак) выдается как результат лексического анализа. Ключевые слова распознаются либо явным выделением непосредственно из текста, либо сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых слов. Если да, то выдается признак соответствующего ключевого слова, если нет - выдается признак идентификатора, а сам идентификатор сохраняется отдельно. Если выделенная лексема принадлежит какому-либо из других классов лексем (число, строка и т.д.), то выдается признак класса лексемы, а значение

лексемы сохраняется.



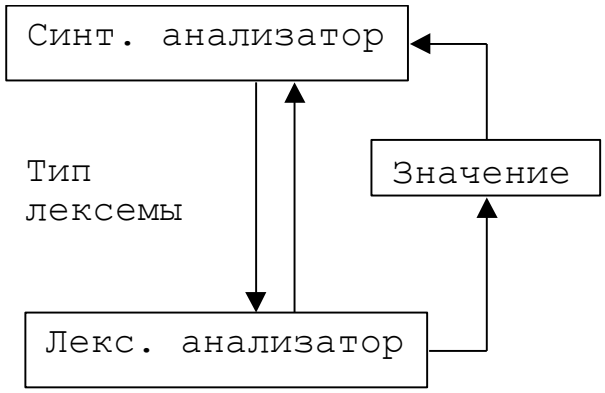


Рис. 2.1 **Рис. 2.2**

Лексический анализатор может работать или как самостоятельная фаза трансляции, или как подпрограмма, работающая по принципу "дай лексему". В первом случае (рис. 2.1) выходом лексического анализатора является файл лексем, во втором (рис. 2.2) лексема выдается при каждом обращении к лексическому анализатору (при этом, как правило, тип лексемы возвращается как значение функции "лексический анализатор", а значение передается через глобальную переменную). С точки зрения формирования значений лексем, принадлежащих классам лексем, лексический анализатор может либо просто выдавать значение каждой лексемы и в этом случае построение таблиц переносится на более поздние фазы, либо он может самостоятельно строить таблицы объектов (идентификаторов, строк, чисел и т.д.). В этом случае в качестве значения лексемы выдается указатель на вход в соответствующую таблицу.

Работа лексического анализатора описывается формализмом конечных автоматов. Однако, непосредственное описание конечного автомата неудобно практически. Поэтому для описания лексических анализаторов, как правило, используют либо формализм регулярных выражений, либо формализм контекстно свободных грамматик, а именно подкласса автоматных, или регулярных, грамматик. Все три формализма (конечных автоматов, регулярных выражений и автоматных грамматик) имеют одинаковую выразительную мощность. По описанию лексического анализатора в виде регулярного выражения или автоматной грамматики строится конечный автомат, распознающий соответствующий язык.

2.1. Регулярные множества и регулярные выражения

Пусть T - конечный алфавит. *Регулярное множество* в алфавите T определяется рекурсивно следующим образом (знаком '<-' будем обозначать принадлежность множеству, знаком '<=' включение):

(1) {} (пустое множество) - регулярное множество в алфавите T;

(2) {a} - регулярное множество в алфавите T для каждого a<-T;

(3) {*e*} - регулярное множество в алфавите T (*e* - пустая цепочка);

(4) если P и Q - регулярные множества в алфавите T, то таковы же и множества

(а) P **U** Q (объединение),

(б) PQ (конкатенация, т.е. множество pq, p<-P, q<-Q),

(в) P* (итерация: P*={*e*} **U** P **U** PP **U**...;

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

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

Приведенное выше определение регулярного множества одновременно определяет и форму его записи, которую будем называть *регулярным выражением*. Для сокращенного обозначения выражения PP* будем пользоваться записью P+ и там, где это необходимо, будем использовать скобки. В этой записи наивысшим приоритетом обладает операция *, затем конкатенация и, наконец, операция **U**, для записи которой иногда будем использовать значок '|'. Так, 0|10* означает (0|(1(0*))). Кроме того, мы будем использовать запись вида

$d_1 = r_1$
$d_2 = r_2$
.....
$d_n = r_n$

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

Пример 2.1. Несколько примеров регулярных выражений и обозначаемых ими множеств

Идентификатор - это регулярное выражение

```
Идентификатор = Буква (Буква|Цифра)*
Буква = {a,b,...,z}
Цифра = {0,1,...,9}
```

Число в десятичной записи - это регулярное выражение

```
Целое = Цифра+
Дробная_часть = . Целое | e
Спепень = ( E ( + | - | e ) Целое ) | e
Число = Целое Дробная_часть Степень
```

Ясно, что для каждого регулярного множества можно найти по крайней мере одно регулярное выражение, обозначающее это множество. И обратно: для каждого регулярного выражения можно построить регулярное множество, обозначаемое этим выражением. Для каждого регулярного множества существует бесконечно много обозначающих его регулярных выражений. Будем говорить, что два регулярных выражения *равны*, если они обозначают одно и то же множество.

2.2. 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*;
(3) D - *функция переходов*, отображающая множество QxTU{*e*} во множество подмножеств множества Q и определяющая поведение управляющего устройства;
(4) q_0<-Q - *начальное состояние* управляющего устройства;
(5) F<=Q - множество *заключительных состояний*.

Детерминированный конечный автомат (ДКА) - это пятерка M=<Q,T,D,q_0,F>, где

(1) Q - конечное *множество состояний*;
(2) T - конечное *множество допустимых входных символов*;
(3) D - *функция переходов*, отображающая множества QxT в множество Q и определяющая поведение управляющего устройства;
(4) q_0<-Q - *начальное состояние* управляющего устройства;
(5) F<=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 сдвига входной головки на одну ячейку вправо (рис. 2.3).

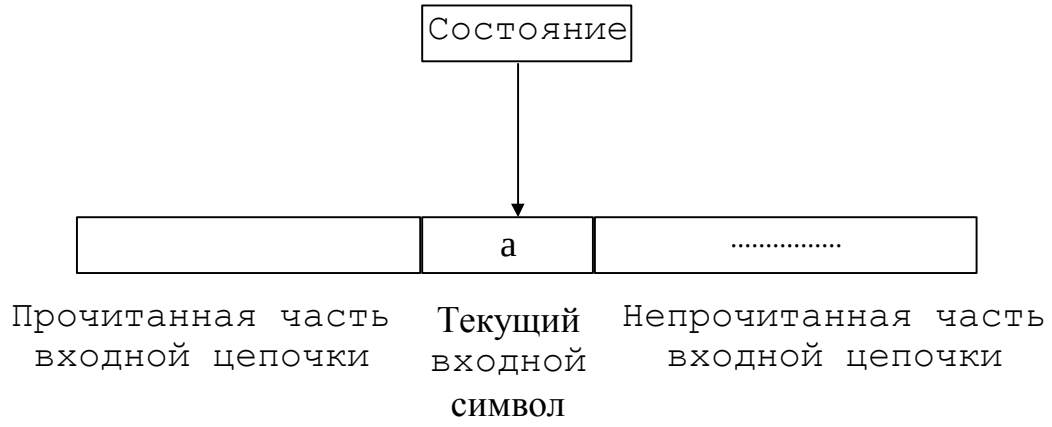


Рис. 2.3

Текущее состояние управляющего устройства, символ под головкой и цепочка символов вправо от головки называются *конфигурацией* автомата. Конфигурация (q_0,w) называется *начальной*, а пара (q,*e*), где q<-F, называется *заключительной* (или *допускающей*).

Такт автомата M представляется бинарным отношением |-, определенным на конфигурациях: отношение имеет место, если есть переход из конфигурации (q_1,w_1) в конфигурацию (q_2,w_2). Для детерминированного конечного автомата всегда w_1=aw_2. Для нетерминированного автомата может быть w_1=w_2, если q_2<-D(q_1,*e*). Отношения |-+ и |-* - это, соответственно, транзитивное и рефлексивно-транзитивное замыкание отношения |-. Говорят, что автомат M *допускает цепочку* w, если (q_0,w)|-*(q,*e*) для некоторого q<-F. *Языком, допускаемым (распознаваемым, определяемым)* автоматом M, (обозначается L(M)), называется множество входных цепочек, допускаемых автоматом M. Т.е.

L(M)={w | w<-T* и (q_0,w)|-*(q,*e*) для некоторого q<-F}

Конечный автомат может быть изображен графически в виде графа, в котором каждому состоянию соответствует вершина, а дуга, помеченная символом a, соединяет две вершины p и q, если функция переходов содержит (q,a)->p. На диаграмме выделяются конечные состояния (в примерах выше двойным контуром).

Пример 2.2. Диаграмма для чисел языка Паскаль приведена на рис. 2.4.

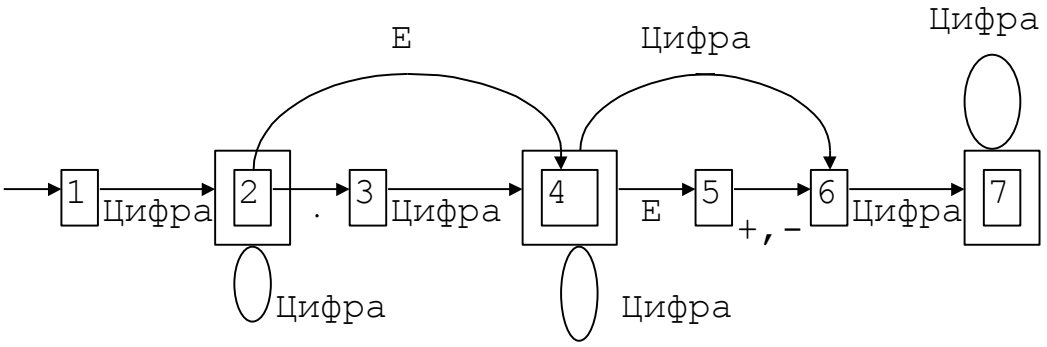


Рис. 2.4

2.3. Построение детерминированного конечного автомата по недерминированному

Алгоритм строит функцию переходов Dtran для детерминированного конечного автомата DFA. Каждое состояние DFA - это некоторое множество состояний недерминированного автомата NFA. DFA моделирует “в параллель” все возможные шаги NFA, которые он может сделать на входной строке. В алгоритме будут использоваться следующие операции:

e-closure(S) - множество состояний NFA, достижимых из некоторого состояния q<-S на *e*-переходах.

move(S,a) - множество состояний NFA, в которые есть переход на входе a для некоторого состояния q<-S.

Прежде, чем увидеть первый входной символ, NFA может находиться в любом состоянии из множества *e*-closure($\{q_0\}$). Пусть S - множество состояний, достижимых NFA на некоторой последовательности входных символов и пусть a - очередной входной символ. Читая a, NFA может перейти в любое из состояний из move(S,a). Если учесть *e*-переходы, NFA может оказаться в любом из состояний из *e*-closure(move(S,a)).

Алгоритм 2.1. Построение детерминированного конечного автомата по недетермнированному

Вначале единственное состояние в Dstates - *e*-closure($\{q_0\}$) и оно не помечено.

```
while (в Dstates есть непомеченное состояние S)
    {пометить S;
     for (каждого входного символа a<-T)
        {R= e-closure(move(S,a));
         if (R !<- Dstates)
            добавить R в Dstates как непомеченное
состояние;
            определить Dtran[S,a]=R;
        }
    }
```

Начальное состояние так построенного автомата DFA - это *e*-closure({q_0}), заключительное - любое такое, которое содержит заключительное состояние NFA.

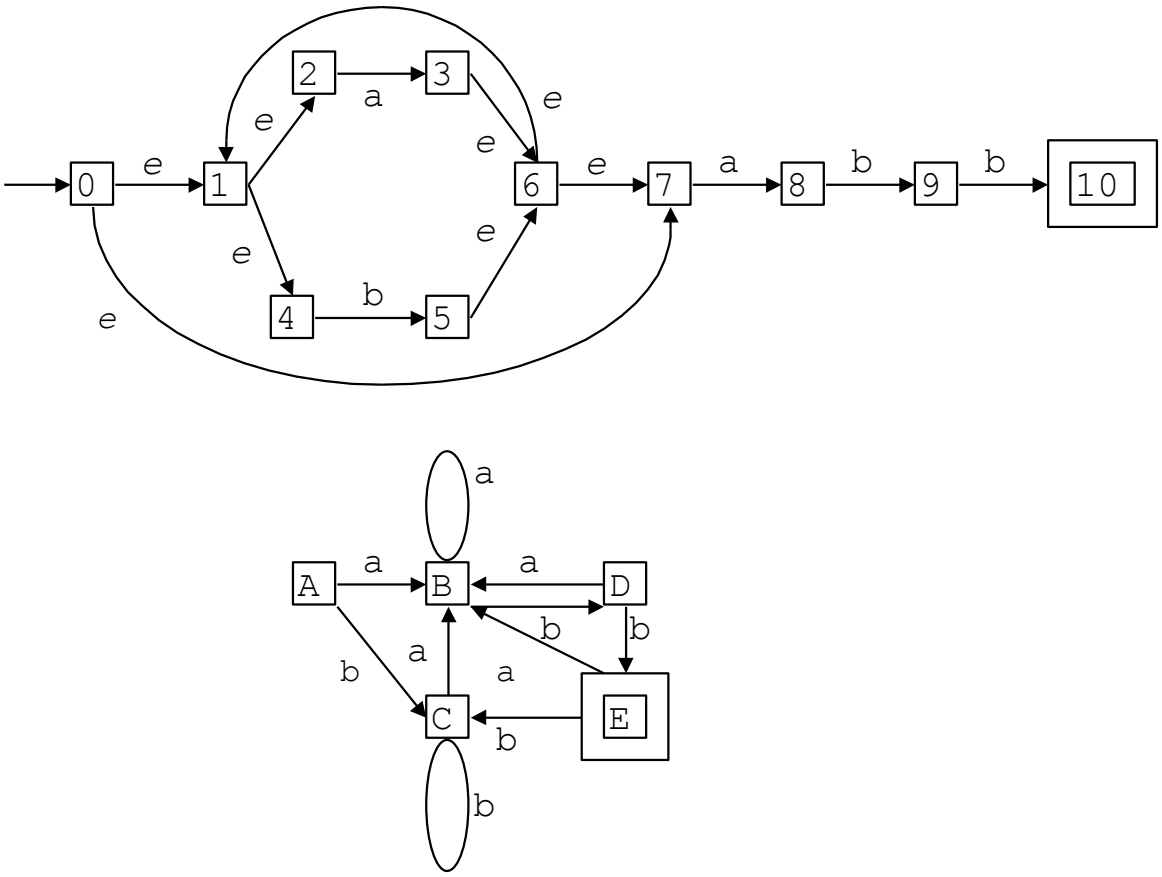


Рис. 2.5

e-closure(S) можно вычислить следующим образом:

```
R=e-closure(S)=S;
while (R!={})
```

```
{Пусть r<-R;
 for (каждого q такого, что q <- D(r,e))
   if (q !<- e-closure(S))
      {добавить q к e-closure(S);
       добавить q к R;
      }
   убрать r из R;
}
```

Пример применения алгортима приведен на рис. 2.5.

2.4. Построение детерминированного конечного автомата по регулярному выражению.

Приведем теперь алгоритм построения детерминированного конечного автомата по регулярному выражению [1]. К регулярному выражению (сокращенно РВ) r добавим маркер конца: (r)#. После построения ДКА для расширенного РВ легко построить ДКА для исходного РВ: все состояния ДКА из которых есть переход в конечное с чтением символа "#", можно считать конечными, а символ "#" и соответствующие переходы удалить.

Представим РВ в виде дерева, листья которого - терминальные символы, а внутренние вершины - операции "." (конкатенации), "U" (объединение), "*" (итерация).

Каждому листу дерева (кроме *e*-листьев) припишем уникальный номер и ссылаться на него будем, с одной стороны, как на позицию в дереве и, с другой стороны, как на позицию символа, соответствующего листу.

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 узлов n, поддеревья которых (т.е. дерево, у которого узел n является корнем) могут породить пустое слово, определим nullable(n)=true, а для остальных узлов false.

узел n	nullable(n)	furstpos(n)	lastpos(n)
лист e	true	{}	{}
лист i	false	{i}	{i}
U /\ u v	nullable(u) or nullable(v)	firstpos(u) U firstpos(v)	lastpos(u) U lastpos(v)
. /\ u v	nullable(u) and nullable(v)	if nullable(u) firstpos(u) U firstpos(v) else firstpos(u)	if nullable(v) lastpos(u) U lastpos(v) else lastpos(v)
* | v	true	firstpos(v)	lastpos(v)

Рис. 2.6

Таблица для вычисления функций nullable, firstpos и followpos приведена на рис. 2.6.

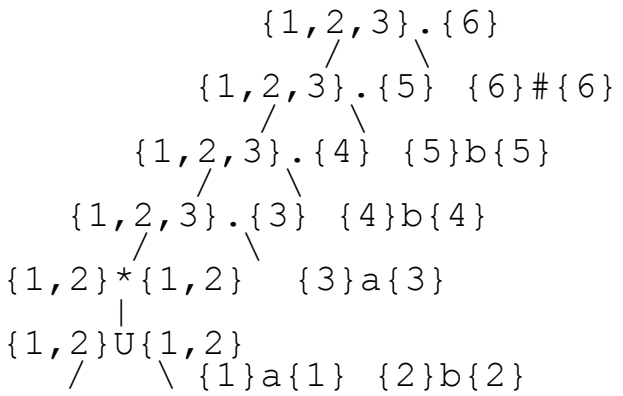


Рис. 2.7

позиция	followpos
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	

Рис. 2.8

Пример 2.3. Функции firstpos и lastpos для выражения (a+b)*abb# приведены на рис. 2.7. Слева от каждой вершины значение firstpos, справа - 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 ...cd..., входящая в язык, описываемый РВ, такая, что i - соответствует этому вхождению c, а 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 "." (конкатенация), a,b - его потомки. Тогда для каждой позиции i, входящей в lastpos(a), добавляем к

множеству значений followpos(i) множество firstpos(b).

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

Для примера 2.3 значения функции followpos приведены на рис. 2.8.

Функция followpos позволит теперь сразу построить детерминированный конечный автомат с помощью следующего алгоритма.

Алгоритм 2.2. Прямое построение ДКА по регулярному выражению.

Будем строить множество состояний автомата Dstates и помечать их. Состояния ДКА соответствуют множествам позиций. Начальным состоянием будет состояние firstpos(root), где root - вершина синтаксического дерева регулярного выражения, конечными - все состояния, содержащие позиции, связанные с символом "#".

Сначала в Dstates имеется только одно непомеченное состояние firstpos(root).

```
while (в Dstates есть непомеченное состояние R)
{пометить R;
   for (каждого входного символа a, такого, что в R
      имеется позиция, которой соответствует a)
      {пусть символ a в R соответствует позициям
          p1,...,pi, и пусть S=U followpos(pi);
                               i
      if (S не пусто и S не принадлежит Dstates)
         добавить непомеченное состояние S в Dstates
            (рис. 2.9);
       Функцию перехода Dtran для R и a определить как
          Dtran(R,a)=S;
      }
   }
```

Для примера 2.3 вначале R={1(a),2(b),3(a)}. Последовательность шагов

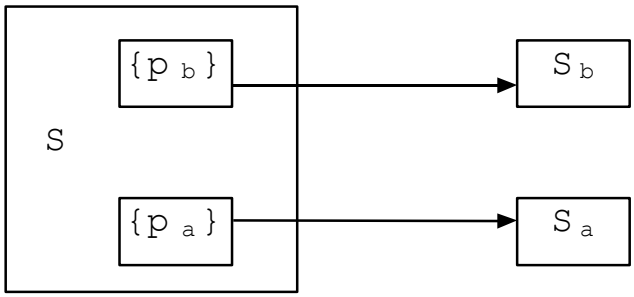


Рис. 2.9

алгоритма приведена на рис. 2.10. В результате будет построен детерминированный конечный автомат, изображенный на рис. 2.11.

Состояния автомата обозначаются как множества позиций, например {1,2,3}, конечное состояние заключено в квадратные скобки [1,2,3,6].

	U followpos(p(a))	U followpos(p(b))
{1,2,3}	{1,2,3,4,}	{1,2,3}
{1,2,3,4}	{1,2,3,4}	{1,2,3,5}
{1,2,3,5}	{1,2,3,4}	{1,2,3,6}
{1,2,3,6}	{1,2,3,4}	{1,2,3}

Рис. 2.10

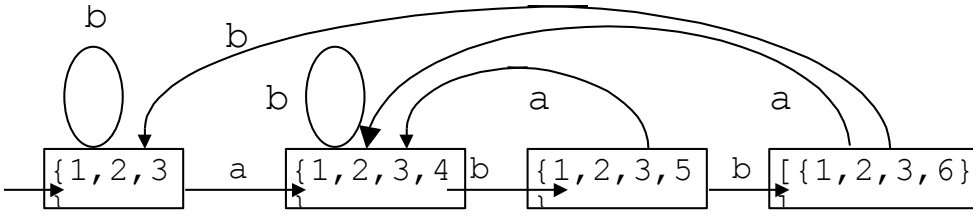


Рис. 2.11

2.5 Построение детерминированного конечного автомата с минимальным числом состояний

Рассмотрим теперь алгоритм построения ДКА с минимальным числом состояний, эквивалентного данному ДКА [2].

Алгоритм 2.3. Построение ДКА с минимальным числом состояний.

Шаг 1. Строим начальное разбиение П множества состояний из двух групп: заключительные состояния Q и остальные Q-F.

Шаг 2. Применяем к П следующую процедуру и получаем новое разбиение $П_{new}$ (рис. 2.12):

```
for (каждой группы G в П)
      {разбиваем G на подгруппы так, чтобы
       состояния s  и t из G оказались в одной
       группе 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 П;
       заменяем G в Пnew на множество всех
```

```
 полученных подгрупп;
}
```

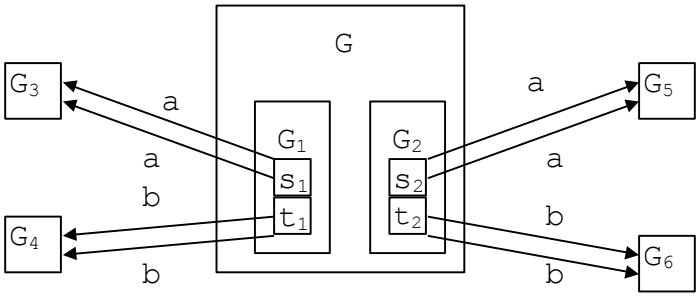


Рис. 2.12

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

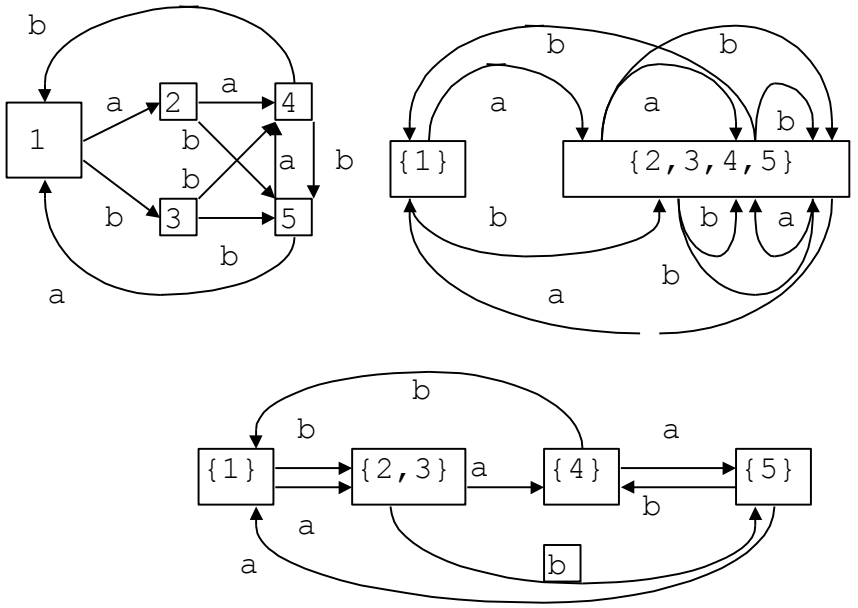


Рис. 2.13

Шаг 4. Выберем по одному состоянию из каждой группы в разбиении $\Pi_{\mathbf{res}}$ в качестве представителя для этой группы. Представители будут состояниями приведенного ДКА M'. Пусть s - представитель. Предположим, что на входе a в M существует переход из s в t. Пусть r - представитель группы t. Тогда M' имеет переход из s в r по a. Пусть начальное состояние M' - представитель группы, содержащей начальное состояние s_0 исходного M, и пусть заключительные состояния M' - представители в F. Отметим, что каждая группа $\Pi_{\mathbf{res}}$ либо состоит только из состояний из F, либо не имеет состояний из F.

Шаг 5. Если M' имеет мертвое состояние, т.е. состояние d, которое не

является допускающим и которое имеет переходы в себя по любому символу, удалим его из M'. Удалим также все состояния, не достижимые из начального.

На рис. 2.13 приведен пример применения алгоритма.

2.6. Программирование лексических анализаторов

Лексический анализатор, как правило, вызывается как подпрограмма. При обращении к ЛА вырабатываются как минимум два результата: тип выбранной лексемы и значение (или указатель на значение) для классов лексем (идентификаторов, чисел, строк и т.д.). Само значение передается, если ЛА не работает с таблицей имен. Если же ЛА сам формирует таблицу имен, то он выдает указатель на имя. Обычно ЛА оформляется как процедура-функция, вырабатывающая тип лексемы и заносящая в некоторую глобальную переменную значение лексемы, если это необходимо. Помимо значения лексемы, эта глобальная переменная может содержать некоторую дополнительную информацию: номер текущей строки, номер символа в строке и другую. Эта информация может использоваться в различных целях, например, для диагностики.

Тело ЛА представляет собой диаграмму переходов соответствующего конечного автомата. Отдельная проблема - анализ ключевых слов. Как правило, ключевые слова - это выделенные идентификаторы. Поэтому возможны два основных способа выделения 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 требуется заглянуть в таблицу 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 и
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 THEN THEN THEN = ELSE; ELSE ELSE = THEN или
DECLARE (ARG1, ARG2, ...., ARGn) ...
```

и только в зависимости от того, что стоит после ")", можно определить, является ли DECLARE именем подпрограммы или объявлением. 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 [2]. Пусть LETTER, DIGIT, BLANK, SLESS - 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;
MAP['<']:=SLESS;
    ........
```

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.

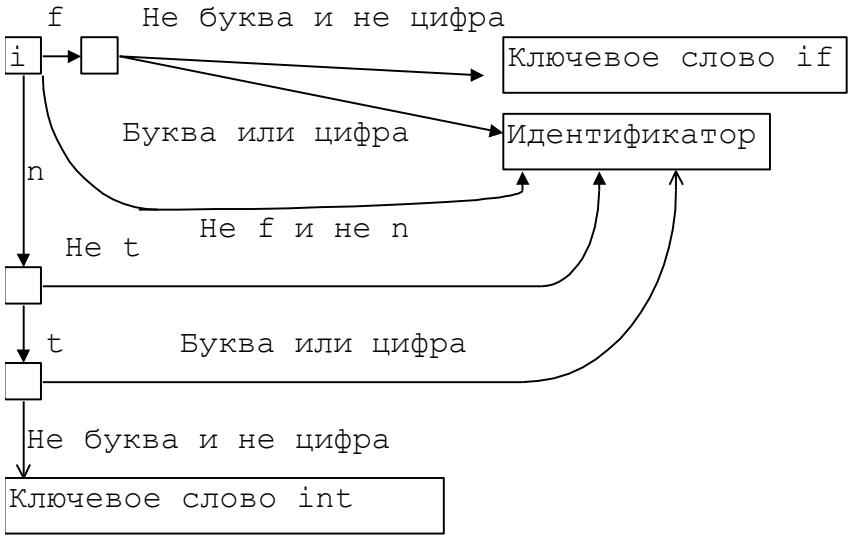


Рис. 2.14

Для этого строится конечный автомат, описывающий множество ключевых слов. На рис. 2.14 приведен фрагмент такого автомата. Рассмотрим пример программирования этого конечного автомата на языке Си, приведенный в [3]:

```
case 'i':
if (cp[0]=='f' &&!(map[cp[1]] & (digit | letter)))
{cp++; return IF;}
if (cp[0]=='n' && cp[1]=='t'
 &&!(map[cp[2]] & (digit | letter)))
{cp+=2; return INT;}
```

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 несколько возможных эффективных схем буферизации.

В первой схеме используется буфер, размер которого - двойная длина блока обмена N (рис. 2.15).

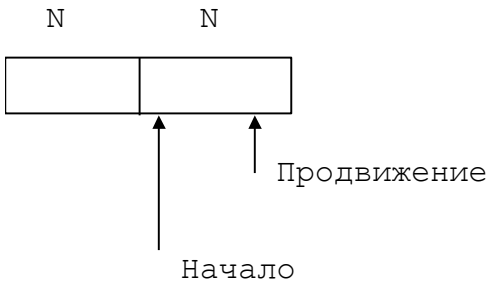


Рис. 2.15

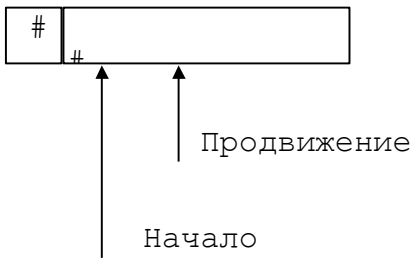


Рис. 2.16

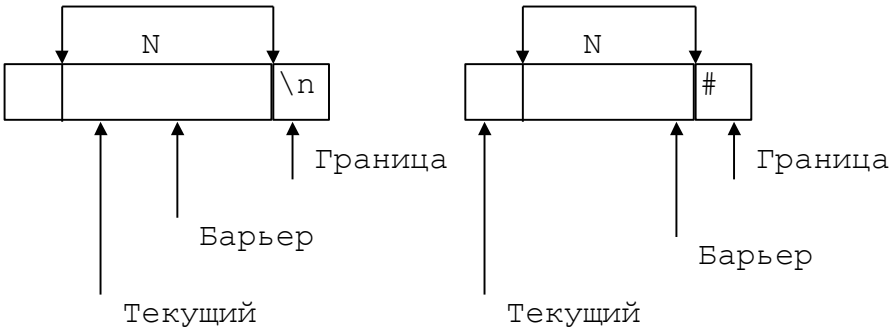
Чтобы не читать каждый символ отдельно, в каждую из половин буфера одной командой чтения считывается N символов. Если на входе осталось меньше N символов, в буфер помещается специальный символ (eof). На буфер указывают два указателя: *продвижение* и *начало*. Между указателями размещается текущая лексема. Вначале они оба указывают на первый символ выделяемой лексемы. Один из них, *продвижение*, продвигается вперед, пока не будет выделена лексема, и 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 '#' (рис. 2.16).

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

В третьей схеме используются три указателя (рис. 2.17). Непросмотренная часть буфера заключена между *текущим* и *границей* (*граница* - это указатель на последний элемент буфера). Анализ очередной лексемы начинается после сканирования незначащих пробелов. Если после этого *текущий* указывает на '#' в конце буфера, делается перезагрузка буфера (предполагается, что '#' не может входить в состав лексемы). *Барьер* выбирается таким образом, чтобы между *барьером* и *границей* всегда помещалась любая лексема. Если начало очередной лексемы оказывается правее *барьера*, то часть буфера от *текущего* до *границы*

переписывается левее буфера и буфер перезагужается. Тем самым начало лексемы конкатенируется с ее концом. Так обрабатывается ситуация, когда граница буфера прошла через лексему.



а) Пока текущий барьер б) после чтения

Рис. 2.17

В результате большинство входных символов обрабатываются непосредственно в буфере. Копируются только идентификаторы и строковые константы в соответствующие таблицы.

2.7. Конструктор лексических анализаторов LEX

Для автоматизации разработки лексических анализаторов было разработано довольно много средств. Как правило, входным языком для них служат либо КС (автоматные) грамматики, либо язык регулярных выражений. Одной из наиболее распространенных систем является LEX, входным языком которого являются регулярные выражения. LEX-программа состоит из трех частей:

```
%START список состояний

Объявления
%%
Правила трансляции
%%
Вспомогательные процедуры
```

Секция START содержит перечисление состояний в которых может находиться анализатор. Секция может остутствовать.

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* любое число (включая 0) повторений регулярного выражения R;

R+ не равное 0 число повторений регулярного выражения R;

R_1/R_2 R_1, если за ним следует R_2;

$R_1|R_2$ 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} повторить R от n до m раз;

{имя} использовать именованное выше регулярное выражение;

(R) группировка.

Правила трансляции LEX программ имеют вид

```
p1 { действие_1 }
p2 { действие_2 }
..............
pn { действие_n }
```

где каждое p_i - регулярное выражение, возможно помеченное именем состояния, 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. Среди действий может быть указано GOTO *состояние*. Выполнение этого действия переводит анализатор в указанное состояние. Если определено некоторое состояние анализатора, то в нем анализируются только выражения, помеченные меткой этого состояния. Все непомеченные действия соответствуют состоянию по умолчанию ‘0’.

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 2.4. На рис. 2.18 приведена LEX-программа.

```
%{ /*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()
{/*процедура, которая помещает лексему,
       на первый символ которой указывает yytext,
       длина которой равна yyleng, в таблицу
```

```
        символов и возвращает указатель на нее*/
}
install_num()
{/*аналогичная процедура для размещения
                  лексемы числа*/
}
```

Рис. 2.18.

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 процедур третьей секции. На рис. 2.18 это процедуры 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 3. Синтаксический анализ

3.1. Основные понятия и определения

Пусть G=<N,T,P,S> - контекстно-свободная грамматика, где N - множество нетерминальных символов, T - множество терминальных символов, P - множество правил вывода и S - аксиома. Будем говорить, что uxv

выводится за один шаг из uAv (и записывать это как uAv=>uxv), если A->x - правило вывода и u и v - произвольные строки из (N U T)*. Если u1=>u2=>...=>un, будем говорить, что из u_1 выводится u_n, и записывать это как u_1=>*u_n. Т.е.:

1) u=>*u для любой строки u,

2) если u=>*v и v=>*w, то u=>*w.

Аналогично, "=>+" означает *выводится за один или более шагов.*

Если дана грамматика G с начальным символом S, отношение =>+ можно использовать для определения L(G) - языка, порожденного G. Строки L(G) могут содержать только терминальные символы G. Строка терминалов w принадлежит L(G) тогда и только тогда, когда S=>+w. Строка w называется *предложением* в G.

Если S=>*u, где u может содержать нетерминалы, то u называется *сентенциальной формой* в G. Предложение - это сентенциальная форма, не содержащая нетерминалов.

Рассмотрим выводы, в которых в любой сентенциальной форме на каждом шаге делается подстановка самого левого нетерминала. Такой вывод называется *левосторонним*. Если S=>*u в процессе левостороннего вывода, то u - *левая сентенциальная форма*. Аналогично определяется правосторонний вывод.

Упорядоченным графом называется пара (V,E), где V обозначает множество вершин, а E - множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид ((v,e_1),(v,e_2),...,(v,e_n)). Этот элемент указывает, что из вершины a выходят n дуг, причем первой из них считается дуга, входящая в вершину e_1, второй - дуга, входящая в вершину e_2, и т.д.

Дерево вывода в грамматике G=<N,T,P,S> - это помеченное упорядоченное дерево, каждая вершина которого помечена символом из множества N U T U {*e*}. Если внутренняя вершина помечена символом A, а ее прямые потомки - символами X_1,..., X_n, то A->$X_1X_2...X_n$ - правило этой грамматики.

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

(1) корень дерева D помечен S;

(2) каждый лист помечен либо a<-T, либо *e*;

(3) каждая внутренняя вершина помечена нетерминалом;

(4) если N - нетерминал, которым помечена внутренняя вершина и $X_1,...,X_n$ - метки ее прямых потомков в указанном порядке, то $N->X_1...X_k$ - правило из множества P.

Автомат с магазинной памятью (сокращенно МП-автомат) - это семерка $P=<Q,T,\Gamma,D,q_0,Z_0,F>$, где

(1) Q - конечное множество *символов состояний*, представляющих всевозможные состояния управляющего устройства;
(2) T - конечный *входной* алфавит;
(3) Г - конечный алфавит *магазинных символов*;
(4) D - *функция переходов* - отображение множества $Qx(T \cup \{e\})x\Gamma$ в множество конечных подмножеств $Qx\Gamma^*$, т.е. $d:Qx(T \cup \{e\})x\Gamma -> \{Qx\Gamma^*\}$;
(5) $q_0<-Q$ - *начальное состояние* управляющего устройства;
(6) $Z_0<-\Gamma$ - символ, находящийся в магазине в начальный момент (*начальный символ*);
(7) $F<=Q$ - множество *заключительных* состояний

Конфигурацией МП-автомата называется тройка $<q,w,u> <- QxT^*x\Gamma^*$, где

(1) q - текущее состояние управляющего устройства;
(2) w - неиспользованная часть входной цепочки; первый символ цепочки w находится под входной головкой; если w *e*, то считается, что вся входная лента прочитана;
(3) u - содержимое магазина; самый левый символ цепочки u считается верхним символом магазина; если u=*e*, то магазин считается пустым.

Такт работы МП-автомата P будем представлять в виде бинарного отношения |-, определенного на конфигурациях. Будем писать

(q,aw,Zu)|-(q',w,vu)

если множество d(q,a,Z) содержит (q',v), где q, q'<-Q, a<-T U {*e*}, w<-T*, Z<-Г, u,v<-Г*.

Начальной конфигурацией МП-автомата P называется конфигурация вида $<q_0,w,Z_0>$, где w<-T*, т.е. управляющее устройство находится в начальном состоянии, входная лента содержит цепочку, которую нужно распознать, а в магазине есть только начальный символ Z_0. Заключительная конфигурация - это конфигурация вида <q,*e*,u>, где q<-F, u<-Г*.

Говорят, что цепочка w *допускается* МП-автоматом P, если $<q_0,w,Z_0>|-^*<q,e,u>$ для некоторых q<-F и u<-Г*. Языком, *определяемым*

(или *допускаемым*) автоматом P (обозначается L(P)), называют множество цепочек, допускаемых автоматом P.

Иногда допустимость определяют несколько иначе: цепочка w допускается МП-автоматом P, если $<q_0,w,Z_0>|-*<q,e,e>$. Эти определения эквивалентны.

3.2. Таблично-управляемый предсказывающий разбор

3.2.1. Алгоритм разбора сверху-вниз

Основная проблема предсказывающего разбора - определение правила вывода, которое нужно применить к нетерминалу. Процесс предсказывающего разбора (сверху-вниз) с точки зрения построения дерева разбора можно проиллюстрировать рис. 3.1. Фрагменты недостроенного дерева соответствуют сентенциальным формам вывода. Вначале дерево состоит только из одной вершины, соответствующей аксиоме S. В этот момент по первому символу входного потока предсказывающий анализатор должен определить правило $S->X_1\ X_2$ ..., 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->a .... Этот процесс затем применяется для следующего самого левого нетерминального символа сентенциальной формы

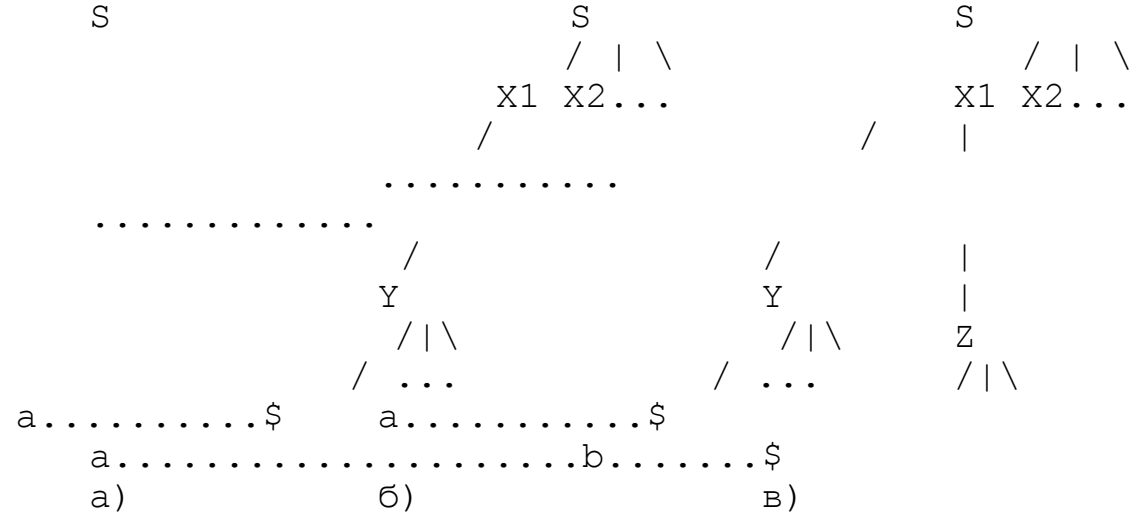


Рис. 3.1

На рис. 3.2 приведена структура предсказывающего анализатора, который определяет очередное правило из таблицы. Такую таблицу можно

построить непосредственно из грамматики.

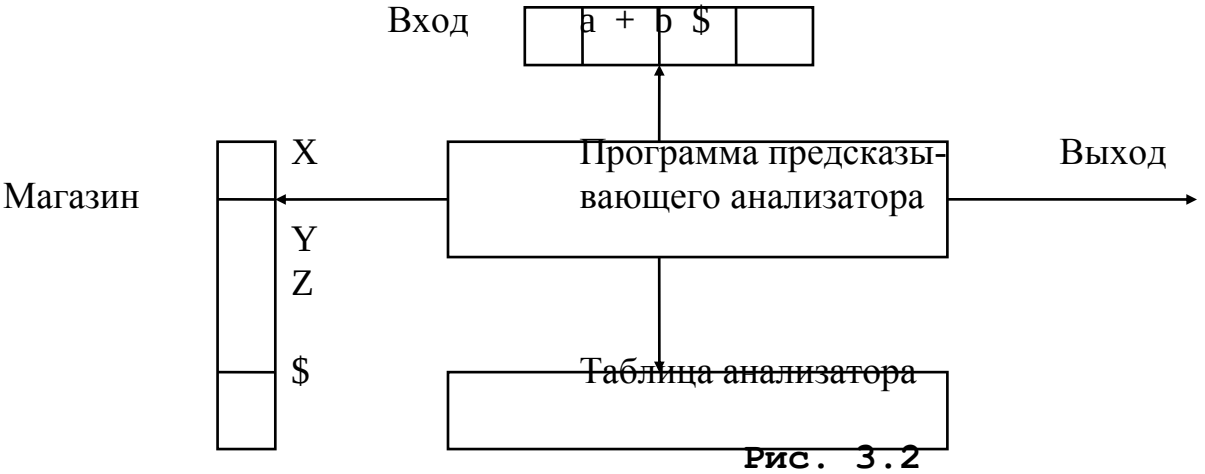


Рис. 3.2

Таблично-управляемый предсказывающий анализатор имеет входной буфер, таблицу анализа и выход. Входной буфер содержит распознаваемую строку, за которой следует $ - правый концевой маркер, признак конца строки. Магазин содержит последовательность символов грамматики с $ на дне. Вначале магазин содержит начальный символ грамматики на верхушке и $ на дне. Таблица анализа - это двумерный массив M[A,a], где A - нетерминал, и a - терминал или символ $.

Анализатор управляется программой, которая работает следующим образом. Она рассматривает X - символ на верхушке магазина и a - текущий входной символ. Эти два символа определяют действие анализатора. Имеются три возможности.

1. Если X=a=$, анализатор останавливается и сообщает об успешном окончании разбора.

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

3. Если X - нетерминал, программа заглядывает в таблицу M[X,a]. По этому входу хранится либо правило для X, либо ошибка. Если, например, M[X,a]={X->UVW}, анализатор заменяет X на верхушке магазина на WVU {на верхушке U}. Будем считать, что анализатор в качестве выхода просто печатает использованные правила вывода. Если M[X,a]=error, анализатор обращается к подпрограмме анализа ошибок.

Поведение анализатора может быть описано в терминах конфигураций автомата разбора.

Алгоритм 3.1. Нерекурсивный предсказывающий анализ.

```
do
   {X=верхний символ магазина;
    if (X <- T|| 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*/
```

Рис. 3.3

Вначале анализатор находится в конфигурации, в которой магазин содержит S$, (S - начальный символ грамматики), во входном буфере w$ (w - входная цепочка), переменная InSym содержит первый символ входной строки. Программа, использующая таблицу анализатора M для осуществления разбора, изображена на рис. 3.3.

Пример 3.1. Рассмотрим грамматику арифметических выражений:

```
E -> T E'
E' -> + T E' | e
T -> F T'
T' -> * F T' | e
F -> ( E ) | id
```

Таблица предсказывающего анализатора для нее изображена на рис. 3.4. Здесь пустые клетки - входы ошибок. Непустые дают правила, по которым делается развертка нетерминала.

Нетер-	Входной символ					
минал	id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'->+TE'			E'->e	E'->e
T	T->FT'			T->FT'		
T'		T'->e	T'->*FT'		T'->e	T'->e
F	F->id			F->(E)		

Рис. 3.4

Магазин	Вход	Выход
$E	d+id*id$	
$E'T	d+id*id$	E->TE'
$E'T'F	d+id*id$	T->FT'
$E'T'id	id+id*id$	F->id
$E'T'	+id*id$	
$E'	+id*id$	T'->e
$E'T+	+id*id$	E'->+TE`
$E'T	id*id$	
$E'T'F	id*id$	T->FT'
$E'T'id	id*id$	F->id
$E'T'	*id$	
$E'T'F*	*id$	T'->*FT'
$E'T'F	id$	
$E'T'id	id$	F->id
$E'T'	$	
$E'	$	T'->e
$	$	E'->e

Рис. 3.5

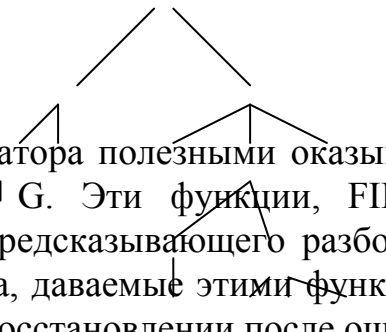


Рис. 3.6

На входе id+id*id предсказывающий анализатор совершает последовательность шагов, изображенную на рис. 3.5. Указатель входа указывает на самый левый символ в колонке Вход. Если внимательно проанализировать действия анализатора, то видно, что он осуществляет левый вывод, т.е. правила применяются в соответствии с левым выводом. За уже просмотренными входными символами следуют символы грамматики в магазине (сверху вниз), что соответствует левым сентенциальным формам вывода.

Дерево разбора приведено на рис. 3.6.

3.2.2. Множества FIRST и FOLLOW.

При построении предсказывающего анализатора полезными оказываются две функции, связанные с грамматикой G. Эти функции, FIRST и FOLLOW, позволяют построить таблицу предсказывающего разбора для G, если, конечно, это возможно. Множества, даваемые этими функциями, могут, кроме того, быть использованы при восстановлении после ошибок.

Если u - любая строка символов грамматики, положим FIRST(u) - множество терминалов, с которых начинаются строки, выводимые из u. Если u=>**e*, то *e* также принадлежит FIRST(u).

Определим FOLLOW(A) для нетерминала A как множество

терминалов a, которые могут появиться непосредственно справа от A в некоторой сентенциальной форме, т.е. множество терминалов a таких, что существует вывод вида S=>*uAav для некоторых u и v. Отметим, что между A и a в процессе вывода могут появиться нетерминальные символы, из которых выводится *e*. Если A может быть самым правым символом некоторой сентенциальной формы, то $ принадлежит FOLLOW(A).

Для построения FIRST(X) для всех символов грамматики X применим следующий алгоритм.

Алгоритм 3.2. Построение множеств FIRST для символов грамматики.

Шаг 1. Если X - терминал, то FIRST(X) - это {X}; если X - нетерминал, полагаем FIRST(X)={}.

Шаг 2. Если имеется правило вывода X->*e*, то добавить *e* к FIRST(X).

Шаг 3. Пока ни к какому множеству FIRST(X) нельзя уже будет добавить новые элементы или *e*:

если X - нетерминал и имеется правило вывода $X->Y_1Y_2...Y_k$, то включить a в FIRST(X), если для некоторого i $a<-FIRST(Y_i)$ и *e* принадлежит всем $FIRST(Y_1),...,FIRST(Y_{i-1})$, т.е. $Y_1...Y_{i-1}=>*e$. Если *e* принадлежит $FIRST(Y_j)$ для всех j=1,2,...,k, то добавить *e* к FIRST(X). Например, все, что принадлежит $FIRST(Y_1)$ принадлежит также и FIRST(X). Если из Y_1 не выводится *e*, то ничего больше не добавляем к FIRST(X), но если $Y_1=>*e$, то добавляем $FIRST(Y_2)$, и т.д.

Теперь FIRST для любой строки $X_1X_2...X_n$ можно вычислить следующим образом.

Шаг 1. Полагаем $FIRST(X_1X_2...X_n)=\{\}$.

Шаг 2. Добавим к $FIRST(X_1X_2...X_n)$ все не *e* символы из $FIRST(X_1)$. Добавим также не *e* символы из $FIRST(X_2)$, если $e<-FIRST(X_1)$,не *e* символы из $FIRST(X_3)$, если *e* принадлежит как $FIRST(X_1)$, так и $FIRST(X_2)$, и т.д. Наконец, добавим *e* к $FIRST(X_1X_2...X_n)$, если $e<-FIRST(X_i)$ для всех i.

Для вычисления FOLLOW(A) для нетерминала A применим алгоритм 3.3.

Алгоритм 3.3. Построение FOLLOW(X) для всех X - нетерминалов грамматики.

Шаг 1. Положить FOLLOW(X)={}.

Шаг 2. Поместить $ в FOLLOW(S), где S - начальный символ и $ - правый концевой маркер.

Шаг 3. Если есть правило вывода A->uBv, то все из FIRST(v), за исключением *e*, добавить к FOLLOW(B).

Шаг 4. Пока ничего нельзя будет добавить ни к какому множеству FOLLOW(X): если есть правило вывода A->uB или A->uBv, где FIRST(v) содержит *e* (т.е. v=>**e*), то все из FOLLOW(A) добавить к FOLLOW(B).

Пример 3.2. Рассмотрим снова грамматику (*). Для нее

```
FIRST(E) =FIRST(T)=FIRST(F)={(,id}
FIRST(E')={+,e}
FIRST(T')={*,e}
FOLLOW(E)=FOLLOW(E')={),$}
FOLLOW(T)=FOLLOW(T')={+,),$}
FOLLOW(F)={+,*,),$}
```

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

На шаге 1 для вычисления множеств FOLLOW в FOLLOW(E) включаем $. На шаге 2, используя правило вывода F->(E), к FOLLOW(E) добавляется также правая скобка. На шаге 3, примененном к правилу E->TE', в FOLLOW(E') включаются $ и правая скобка. Поскольку E'=>**e*, они также попадают в FOLLOW(T). В соответствии с правилом вывода E->TE', на шаге 2 в FOLLOW(T) включается все из FIRST(E'), отличное от *e*.

3.2.3. Конструирование таблиц предсказывающего анализатора

Для конструирования таблиц предсказывающего анализатора по грамматике G может быть использован алгоритм, основанный на следующей идее. Предположим, что A->u - правило вывода грамматики и a<-FIRTS(u). Тогда анализатор делает развертку A по u, если входным символом является a. Трудность возникает, когда u=*e* или u=>**e*. В этом случае нужно развернуть A в u, если текущий входной символ принадлежит FOLLOW(A) или если достигнут $ и $<-FOLLOW(A).

Алгоритм 3.4. Построение таблиц предсказывающего анализатора.

Для каждого правила вывода A->u грамматики выполнить шаги 1 и 2

Шаг 1. Для каждого терминала a из FIRST(u) добавить A->u к M[A,a].

Шаг 2. Если *e*<-FIRST(u), добавить A->u к M[A,b] для каждого терминала b из FOLLOW(A). Если *e*<-FIRST(u) и $<-FOLLOW(A), добавить A->u к M[A,$].

Шаг 3. Положить все неопределенные входы равными error.

Пример 3.3. Применим алгоритм 3.4 к грамматике (*). Поскольку FIRST(TE')=FIRST(T)={(,id}, в соответствии с правилом вывода E->TE' входы M[E,(] и M[E,id] становятся равными E->TE'.

В соответствии с правилом вывода E'->+TE' вход M[E',+] равен E'->+TE'. В соответствии с правилом вывода E'->*e* входы M[E',)] и M[E',$] равны E'->*e*, поскольку FOLLOW(E')={),$}.

Таблица анализа, построенная алгоритмом 3.4, приведена на рис. 3.4.

3.2.4. LL(1)-грамматики

Алгоритм 3.4 для построения таблицы анализа M может быть применен к любой грамматике. Однако для некоторых грамматик M может иметь неоднозначно определенные входы. Например, если грамматика леворекурсивна или неоднозначна, M будет иметь по крайней мере один неоднозначно определенный вход.

Грамматики, для которых таблицы анализа не имеют неоднозначно определенных входов, называются LL(1). Первое L означает сканирование входа слева-направо, второе L означает, что строится левый вывод, 1 - что на каждом шаге для принятия решения используется один символ непросмотренной цепочки. Можно показать, что алгоритм 3.4 для каждой LL(1)-грамматики G строит таблицы, по которым распознаются все цепочки из L(G) и только они.

LL(1)-грамматики имеют несколько отличительных свойств. Неоднозначная или леворекурсивная грамматика не может быть LL(1). Можно также показать, что грамматика G является LL(1) тогда и только тогда, когда для двух правил вида A->u|v выполняется следующее:

1) ни для какого терминала a одновременно из u и v не выводятся строки, начинающиеся с a;

2) только из одной из строк u или v может выводиться пустая строка;

3) если v=>**e*, то из u не выводится никакая строка, начинающаяся с терминала из FOLLOW(A).

Эквивалентным является следующее определение:

КС-грамматика называется LL(1)-грамматикой, если из существования двух левых выводов

(1) S =>* w A u => w v u =>* wx,

(2) S =>* w A u => w z u =>* wy,

для которых FIRST(x)=FIRST(y), вытекает, что v=z. Это означает, что для данной цепочки wAu и первого символа, выводящегося из Au (или $), существует не более одного правила, которое может быть применено к A, чтобы получить вывод какой-нибудь терминальной цепочки, начинающейся с w и продолжающейся этим первым символом.

Язык, для которого можно построить LL(1)-грамматику, называют LL(1)-языком.

Если таблица анализа имеет неоднозначно определенные входы, то грамматика не является LL(1). Примером может служить следующая грамматика:

```
St -> if Ex then St
    | if Ex then St else St
    | Cont
Ex -> ...
```

Эта грамматика неоднозначна, что иллюстрируется рис. 3.7. Поскольку грамматика неоднозначна, она не является LL(1). Проблема, порождает ли грамматика LL-язык, алгоритмически неразрешима.

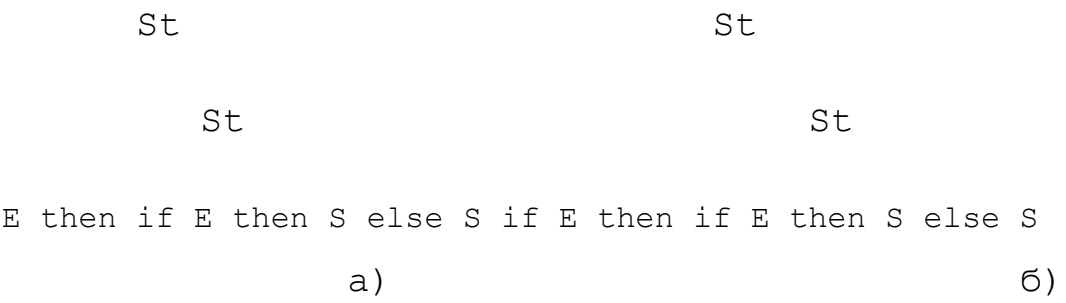


Рис. 3.7

3.2.5. Удаление левой рекурсии

Основная трудность при использовании предсказывающего анализа - это написание такой грамматики для входного языка, чтобы по ней можно было построить предсказывающий анализатор. Иногда с помощью некоторых простых преобразований не LL(1)-грамматику можно привести к LL(1)-виду. Среди этих преобразований наиболее очевидными 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 удаления левой рекурсии и левой факторизации получающаяся грамматика может стать трудно понимаемой.

Грамматика леворекурсивна, если в ней имеется нетерминал A такой, что существует вывод A=>+Au для некоторой строки u. Леворекурсивные грамматики не могут анализироваться методами сверху-вниз, поэтому необходимо удаление левой рекурсии.

Непосредственную левую рекурсию, т.е. рекурсию вида A->Au, можно удалить следующим способом. Сначала группируем A-правила:

$A \to Au_1 \mid Au_2 \mid \ldots \mid Au_m \mid v_1 \mid v_2 \mid \ldots \mid v_n$

где никакая из строк v_i не начинается с A. Затем заменяем A-правила на

$A \to v_1A' \mid v_2A' \mid \ldots \mid v_nA'$
$A' \to u_1A' \mid u_2A' \mid \ldots \mid u_mA' \mid \varepsilon$

Нетерминал A порождает те же строки, 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 шагов. Приведенный ниже алгоритм 3.5 позволяет удалить все левые рекурсии из грамматики.

Алгоритм 3.5. Удаление левой рекурсии.

```
Шаг 1. Упорядочиваем нетерминалы в произвольном
порядке.
Шаг 2. for (i=1;i<=n;i++)
      {for (j=1;j<=i-1;j++)
      {/*По индукции v1,...,vk не имеют ссылок на Ai в
начале*/
            пусть Aj->v1 | v2 | ... | vk - все текущие
правила
              для Aj;
            заменить все правила вида Ai->Aju на
правила
              Ai->v1u | v2u | ... | vku;
      }
         удалить непосредственную левую рекурсию
         в правилах для Ai;
      }
```

После (i-1)-й итерации внешнего цикла на шаге 2 для любого правила вида $A_k \to A_lu$, где k<i, выполняется l>k. В результате на следующей итерации (по i) внутренний цикл (по j) последовательно увеличивает нижнюю границу по m в любом правиле $A_i \to A_mu$, пока не будет m>=i. Затем, удаляя непосредственную левую рекурсию для A_i-правил, делаем m больше i.

Алгоритм 3.5 применим, если грамматика не имеет циклов (выводов

вида A=>+A) и *e*-правил (правил вида A->*e*). Как циклы, так и *e*-правила могут быть удалены предварительно. Получающаяся грамматика без левой рекурсии может иметь *e*-правила.

3.2.6. Левая факторизация

Основная идея левой факторизации в том, что, когда неясно, какую из двух альтернатив надо использовать для развертки нетерминала A, нужно переделать A-правила так, чтобы отложить решение до тех пор, пока не будет досточно информации, чтобы принять правильное решение.

Если A->uv_1 | uv_2 - два A-правила и входная строка начинается с непустой строки, выводимой из u, мы не знаем, разворачивать ли по uv_1 или по uv_2. Однако можно отложить решение, развернув A->uA'. Тогда после анализа того, что выводимо из u, можно развернуть A'->v_1 или A'->v_2. Левофакторизованные правила принимают вид

```
A -> u A'
A' -> v1 | v2
```

Алгоритм 3.6. Левая факторизация грамматики.

Для каждого нетерминала A ищем самый длинный префикс u, общий для двух или более его альтернатив. Если u#*e*, т.е. существует нетривиальный общий префикс, заменяем все A-правила A->uv_1 | uv_2 | ... | uv_n | z, где z - все альтернативы, не начинающиеся с u, на

```
A -> uA' | z
A' -> v1 | v2 | ... | vn
```

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

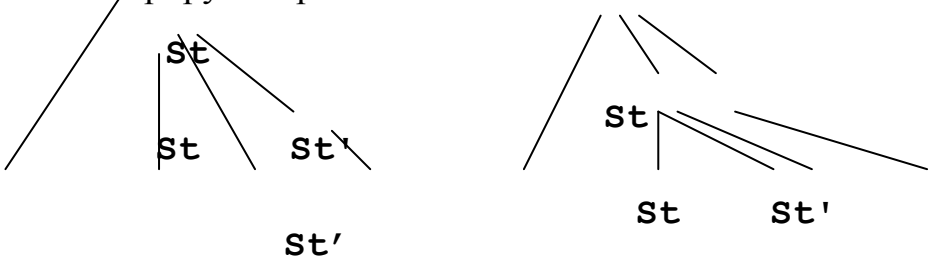
Пример 3.4. Рассмотрим вновь грамматику условных операторов:

```
St -> if Ex then St
    | if Ex then St else St
    | Cont
Ex -> ...
```

После левой факторизации грамматика принимает вид

```
St  -> if Ex then St St'
     | Cont
St' ->  else St | e
Ex  -> ...
```

К сожалению, грамматика остается неоднозначной, а значит, и не LL(1), что иллюстрируется рис. 3.8.



if E then if E then S else S **if E then if E then S S' else S**

а) **б)**

Рис. 3.8

3.2.7. Рекурсивный спуск

Выше был рассмотрен таблично-управляемый вариант предсказывающего анализа, когда магазин явно использовался в процессе работы анализатора. Можно предложить другой вариант предсказывающего анализатра, когда каждому нетерминалу сопоставляется, вообще говоря, рекурсивная процедура и магазин образуется неявно при вызовах этих процедур. Процедуры рекурсивного спуска могут быть записаны, как это изображено на рис. 3.9. В процедуре N для случая, когда имеется альтернатива $N->u_i->^*e$ (не может быть более одной альтернативы, из которой выводится *e*!), приведены два варианта 1.1 и 1.2. В варианте 1.1 делается проверка, принадлежит ли следующий входной символ FOLLOW(N). Если нет - выдается ошибка. Во втором варианте этого не делается, так что анализ ошибки откладывается на процедуру, вызвавшую N.

```
void N()/*N -> u1 | u2 | ... |uk*/
   {if (InSym<-FIRST(ui))//только одному!
      if (parse(ui))
         result("N->ui");
      else error();
    else
//Вариант 1:
   if (имеется правило N->ui =>* e)
//Вариант 1.1
      if (InSym<-FOLLOW(N))
         result("N->ui");
      else error();
```

```
 //Конец варианта 1.1


//Вариант 1.2:
      result("N->ui");
//Конец варианта 1.2
//Конец варианта 1
   Вариант 2:
   if (нет правила N->ui =>*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);
}
```

Рис. 3.9

3.2.8. Диаграммы переходов для рекурсивного спуска

Как правило, непосредственное программирование рекурсивного спуска из грамматики приводит к большому числу процедур. Число этих процедур можно уменьшить, заменив в некоторых правилах рекурсию циклом. Для этого можно воспользоваться диаграммой переходов грамматики, которая строится следующим образом.

Пусть имеется LL(1)-грамматика. Тогда для каждого нетерминала построение диаграммы включает следующие шаги.

Шаг 1. Вводим начальное и заключительное состояния.

Шаг 2. Для каждого правила вывода $A->X_1X_2...X_n$ строим путь из начального в конечное состояние с дугами, помеченными $X_1,...,X_n$. Если анализатор, построенный по диаграмме переходов, оказывается в

состоянии s и дуга, помеченная терминалом a, ведет в состояние t, то если очередной входной символ равен a, анализатор продвигает вход на одну позицию вправо и переходит в состояние t. Если же дуга помечена нетерминалом A, анализатор входит в начальное состояние для A без продвижения входа. После того как он достигает заключительного состояния для A, он переходит в состояние t, что означает "чтение" A из входа при переходе из состояния s в состояние t. Наконец, если есть дуга из s в t, помеченная *e*, то анализатор переходит из s в t, не читая входа.

Если следовать программе рекурсивного спуска, то переход по *e* должен всегда выбираться в качестве последней альтернативы.

Диаграммы переходов могут быть упрощены подстановкой одной в другую. Рассмотрим, например, диаграммы для арифметических выражений на рис. 3.10.

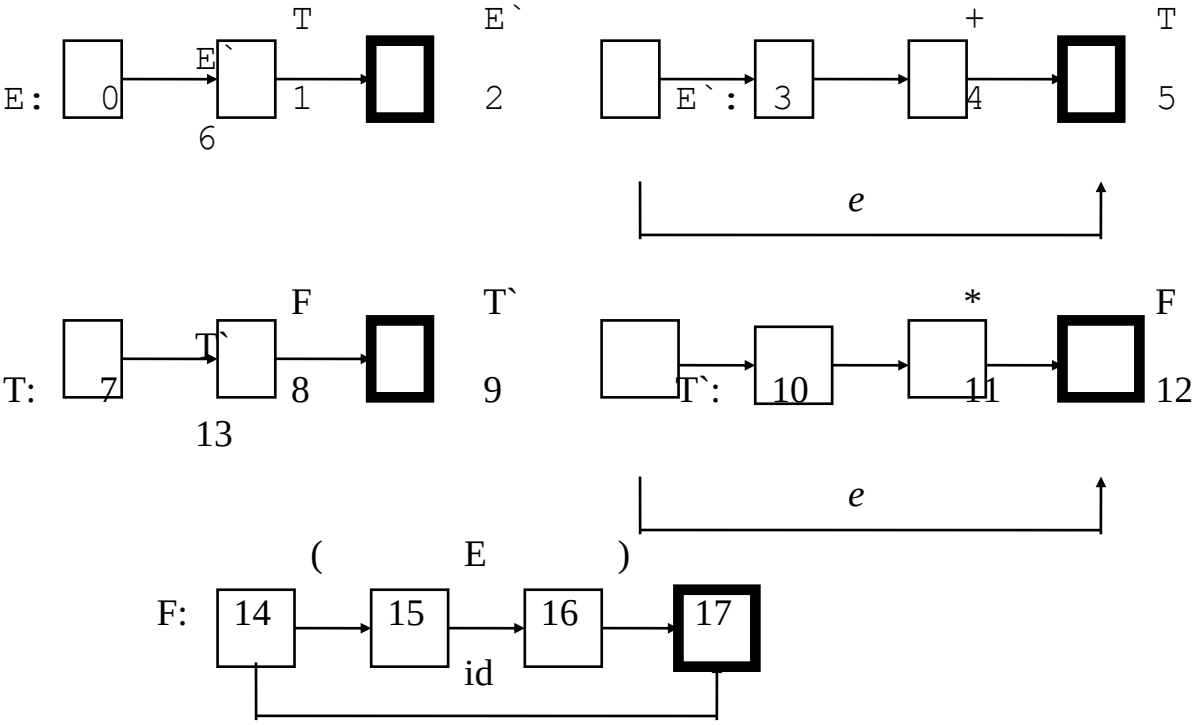
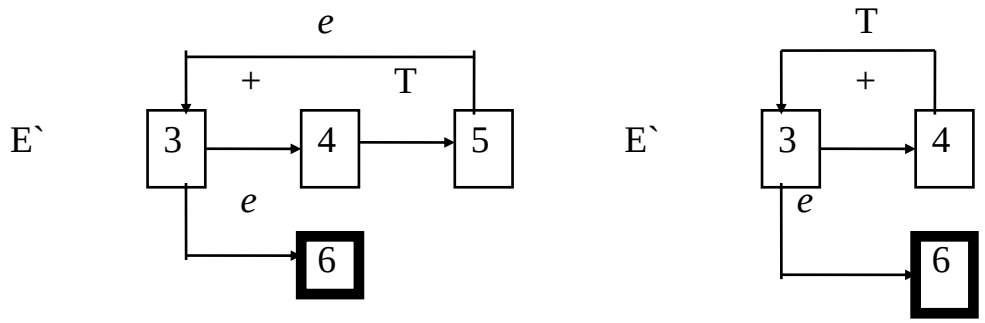


Рис. 3.10



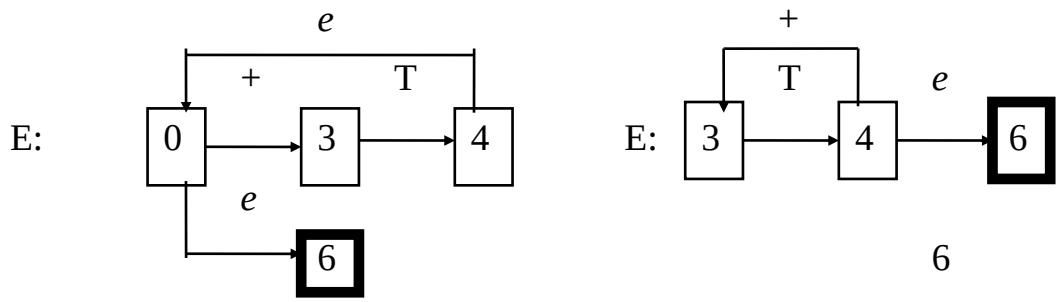


Рис. 3.11

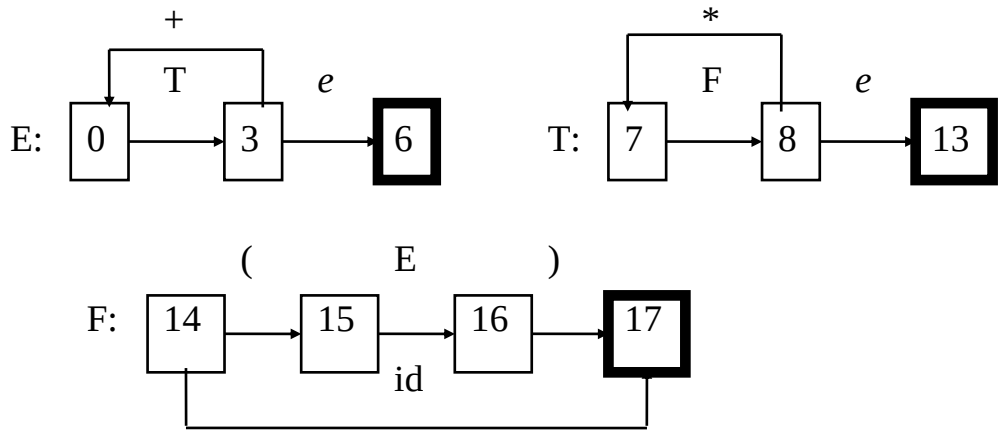


Рис. 3.12

На рис 3.11 приведена эквивалентная диаграмма переходов для E'. Можно подставить диаграмму для E' рис.3.11 в диаграмму для E рис. 3.10. Получится диаграмма рис. 3.11 для E. Наконец, видно, что в этой диаграмме нулевая и четвертая вершины эквивалентны и их можно слить. Так же можно поступить с диаграммами для T и T'. В результате получится набор диаграмм рис. 3.12.

Такое преобразование эквивалентно описанию грамматики расширенными формулами Бэкуса-Наура, в которых помимо собственно рекурсивных определений допускаются описания повторений. При программировании рекурсивного спуска такая диаграмма для E записывается очевидным образом:

```
procedure E; repeat T; until InSym<>PLUS;
```

3.2.9. 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 N и для него нет правила, соответствующего входному символу, то сканируем вход до тех пор, пока не встретим символ либо из FIRST(N), либо из FOLLOW(N). В первом случае разворачиваем N по соответствующему правилу, во втором - удаляем N из магазина.

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

3.3. Разбор снизу-вверх типа сдвиг-свертка

3.3.1. Основа

В процессе разбора снизу-вверх типа сдвиг-свертка строится дерево разбора входной строки, начиная с листьев (снизу) к корню (вверх). Этот процесс можно рассматривать как "свертку" строки w к начальному символу грамматики. На каждом шаге свертки подстрока, которую можно сопоставить правой части некоторого правила вывода, заменяется символом левой части этого правила вывода, и если на каждом шаге выбирается правильная подстрока, то в обратном порядке прослеживается правосторонний вывод (рис. 3.13).

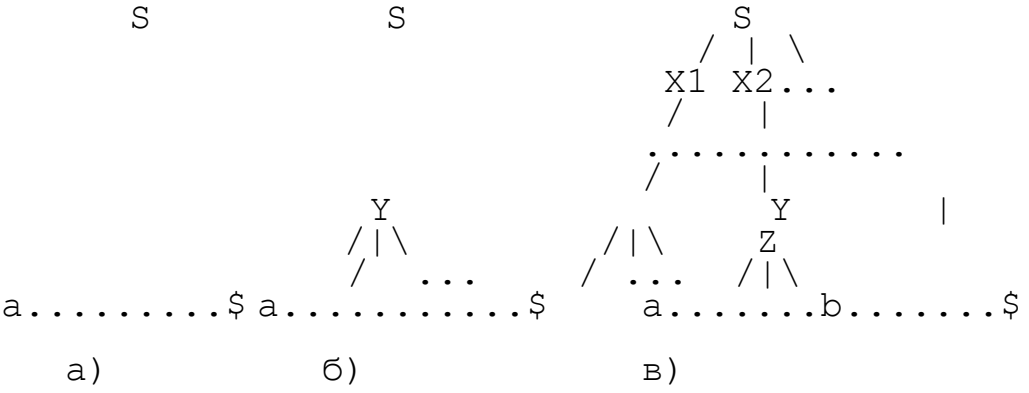


Рис. 3.13

Пример 3.5. Рассмотрим грамматику арифметических выражений, приведенную на рис. 3.14 а). Строка a+b*c может быть сведена к S, как показано на рис. 3.14.б). Дерево этой строки приведено на рис. 3.14 в).

В строке a+b*c ищется подстрока, которую можно сопоставить с правой частью некоторого правила вывода. Этому удовлетворяют подстроки a, b и c. Если выбрать самое левое a и заменить его на F - левую часть правила F->id, то получим строку F+b*c. Теперь правой части того же правила можно сопоставить подстроки b и c. Эти свертки представляют собой в обратном порядке правосторонний вывод:

```
E->E+T->E+T*F->E+T*c->E+F*c->E+b*c->T+b*c->F+b*c-
>a+b*c
```

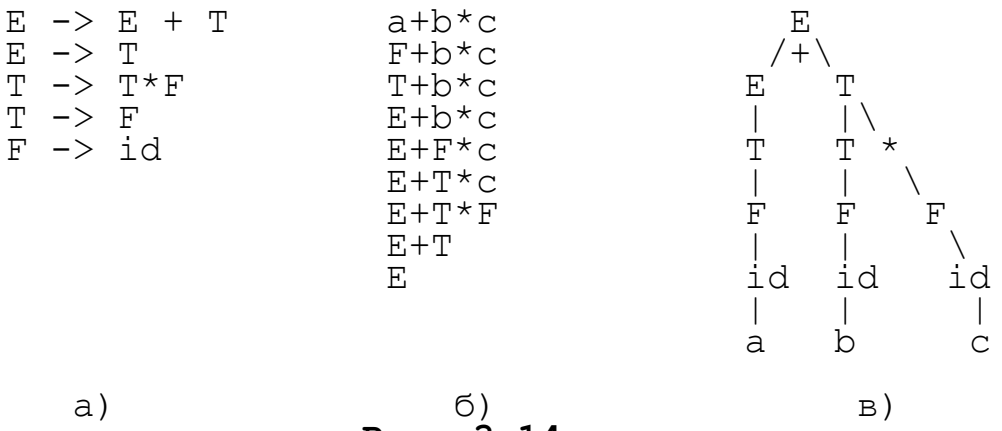


Рис. 3.14

Подстрока 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* строки. В приведенном выше выводе основы выделены. Самая левая подстрока, которая сопоставляется правой части некоторого правила вывода A->v, не обязательно является основой, поскольку свертка по правилу A->v может дать строку, которая не может быть сведена к аксиоме. Если, скажем, в примере 3.5 заменить a на F и b на F, то получим строку F+F*c, которая не может быть сведена к S.

Формально, основа правой сентенциальной формы z - это правило вывода A->v и позиция в z, в которой может быть найдена строка v такие, что в результате замены v на A получается предыдущая сентенциальная форма в правостороннем выводе z. Таким образом, если S=>*uAw=>uvw, то A->v в позиции, следующей за u, это основа строки uvw. Строка w справа от основы содержит только терминальные символы. Вообще говоря, грамматика может быть неоднозначной, поэтому не единственным может быть правосторонний вывод uvw 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 разбираемой строки w. Если w - слово в рассматриваемой грамматике, то $w=Z_n$, n-я правая сентенциальная форма еще неизвестного правого вывода

$$S = Z_0 => Z_1 => Z_2 => \ldots => Z_{n-1} => Z_n = w.$$

Чтобы восстановить этот вывод в обратном порядке, выделяем основу V_n в Z_n и заменяем V_n на левую часть некоторого правила вывода A_n -> V_n, получая (n-1)-ю правую сентенциальную форму Z_{n-1}. Затем повторяем этот

процесс, т.е. выделяем основу V_{n-1} в Z_{n-1} и сворачиваем эту основу, получая правую сентенциальную форму Z_{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.

3.3.2. LR(k)-анализаторы

В названии LR(k) символ L означает, что разбор осуществляется слева-направо, R - что строится правый вывод в обратном порядке, k - число входных символов, на которые заглядывает вперед анализатор при разборе. Когда k опущено, имеют в виду 1.

LR-анализ привлекателен по нескольким причинам:

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

Схематически структура LR-анализатора изображена на рис. 3.15. Он состоит из входа, выхода, магазина, управляющей программы и таблицы анализа, которая имеет две части - *действий* и *переходов*. Управляющая программа одна и та же для всех анализаторов, разные анализаторы различаются таблицами анализа. Программа анализатора читает символы из входного буфера по одному за шаг. В процессе анализа используется магазин, в котором хранятся строки вида $S_0X_1S_1X_2S_2...X_mS_m$ (S_m - верхушка магазина). Каждый X_i - символ грамматики (терминальный или нетерминальный), а S_i - символ, называемый *состоянием*. Каждый символ состояния выражает информацию, содержащуюся в магазине ниже него, а комбинация символа состояния на верхушке магазина и текущего входного символа используется для индексации таблицы анализа и определяет решение о сдвиге или свертке. В реализации символы

состояния не обязательно должны размещаться в магазине. Однако их использование удобно для упрощения понимания поведения LR-анализатора.

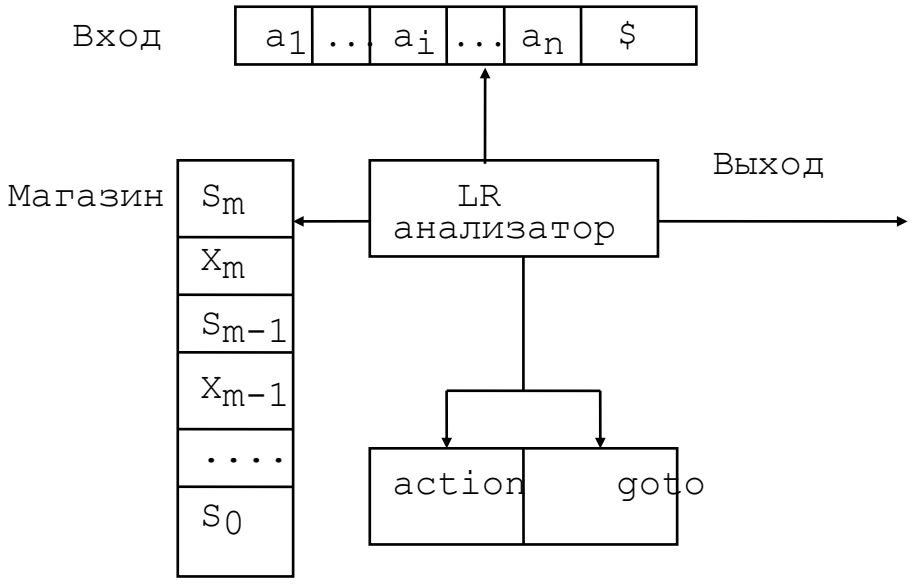


Рис. 3.15

Таблица анализа состоит из двух частей: действия (action) и переходов (goto). Начальное состояние этого ДКА - это состояние, помещенное на верхушку магазина LR-анализатора в начале работы.

Конфигурация-LR анализатора - это пара, первая компонента которой - содержимое магазина, а вторая - непросмотренный вход:

(S0 X1 S1 X2 S2 ... Xm Sm, ai ai+1 ... an $)

Эта конфигурация соответствует правой сентенциальной форме

$X_1\ X_2\ \ldots\ X_m\ a_i\ a_{i+1}\ \ldots\ a_n$

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.

Очередной шаг анализатора определяется текущим входным символом a_i и символом состояния на верхушке магазина S_m. Элемент таблицы действий action[S_m,a_i] для состояния S_m и входа a_i, может иметь одно их четырех значений:

1) shift S, сдвиг, где S - состояние,

2) reduce A->w, свертка по правилу грамматики A -> w,

3) accept, допуск,

4) error, ошибка.

Конфигурации, получающиеся после каждого из четырех типов шагов, следующие

1. Если action[S_m,a_i]=shift S, то анализатор выполняет шаг сдвига, переходя в конфигурацию

$(S_0\ X_1\ S_1\ X_2\ S_2\ \ldots\ X_m\ S_m\ a_i\ S,\ a_{i+1}\ \ldots\ a_n\ \$)$

В магазин помещаются как входной символ a_i, так и следующее состояние S, определяемое action[S_m,a_i]. Текущим входным символом становится a_{i+1}.

2. Если action[S_m,a_i]=reduce A->w, то анализатор выполняет свертку, переходя в конфигурацию

$(S_0\ X_1\ S_1\ X_2\ S_2\ \ldots\ X_{m-r}\ S_{m-r}\ A\ S,\ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$

где S=goto[S_{m-r},A] и r - длина w, правой части правила вывода. Функция goto таблицы анализа, построенная по грамматике G, - это функция переходов детерминированного конечного автомата, распознающего активные префиксы G. 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 как A - левую часть правила вывода, так и S - содержимое таблицы goto[S_{m-r},A]. На шаге свертки текущий входной символ не меняется. Для LR-анализаторов X_{m-r+1} ... X_m - последовательность символов грамматики, удаляемых из магазина, всегда соответствует w - правой части правила вывода, по которому делается свертка.

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

3. Если action[S_m,a_i]=accept, то разбор завершен.

4. Если action[S_m,a_i]=error, анализатор обнаружил ошибку, то выполняются действия по диагностике и восстановлению.

Ниже приведен алгоритм LR-анализа. Все LR-анализаторы ведут себя одинаково. Разница между ними заключается в различном содержании таблиц действий и переходов.

Алгоритм 3.7. Алгоритм LR-анализа.

```
while (true)
   {Пусть S - состояние на верхушке магазина;
    if (action[S,InSym]=="shift S'")
      {поместить InSym и затем S' на верхушку
      магазина;
        прочитать в InSym следующий входной символ;
      }
   else if (action[S,InSym]=="reduce N->w")
         {удалить из магазина 2*|w| символов;
      пусть теперь на верхушке магазина состояние S';
           поместить на верхушку магазина N, а
              затем состояние goto[S',N];
           вывести правило N->w;
         }
      else if (action[S,InSym]=="accept")
            {result("success");
             break;
            }
          else {result(error());
               break;
                }
   }
```

Вначале в магазине помещено начальное состояние S_0, а в буфере w$, InSym содержит первый символ w$; Анализатор выполняет приведенную программу до тех пор, пока будет достигнуто либо состояние accept, либо состояние error.

Пример 3.6. На рис. 3.16 изображены функции action и goto LR-таблиц для грамматики арифметических выражений с бинарными операциями + и * примера 3.5. Здесь S_i означает сдвиг и помещение в магазин состояния i, R_j - свертку по правилу номер j, acc - допуск, пустая клетка - ошибку.

На входе id+id*id последовательность состояний магазина и входа показаны на рис. 3.17. Например, в первой строке LR-анализатор находится в нулевом состоянии и читает первый входной символ id. Действие S6 в нулевой строке и столбце id в поле action рис. 3.17 означает сдвиг и помещение S6 на верхушку магазина. Это и изображено во второй строке: первый символ id и символ состояния S6 помещаются в магазин и id удаляется из входной строки.

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. 3.16

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

Рис. 3.17

Текущим входным символом становится +, и действием в состоянии 6 на вход + является свертка по F->id. Из магазина удаляются два символа (один символ состояния и один символ грамматики). Теперь 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.

3.3.3. LR-грамматики

Грамматики, для которых можно построить таблицу LR-разбора, называются LR-грамматиками. Есть КС-грамматики, не являющиеся LR-грамматиками, однако практически для описания языков программирования достаточно класса LR.

Чтобы грамматика была LR, анализатор, работающий слева-направо по типу сдвиг-свертка, должен уметь распознавать основы на верхушке магазина. Выделение основы осуществляется конечным автоматом, читающим содержимое магазина от дна к верхушке. Состояние автомата после прочтения содержимого магазина и текущий входной символ определяют очередное действие автомата. Функцией переходов этого конечного автомата является таблица 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.

Для принятия решения о сдвиге или свертке анализатор просматривает очередные k входных символов. Практический интерес представляют случаи k=0 и k=1. Например, в таблице действий рис. 3.16 используется один символ. Грамматика, которая может быть проанализирована LR анализатором, заглядывая на k входных символов на каждом шаге, называется *LR(k)-грамматикой*.

Можно дать другое определение LR(k)-грамматики. *Пополненной* грамматикой для данной грамматики G называется КС-грамматика, в которой введена новая аксиома S' и правило вывода S'->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 осуществляет свертку по правилу S'->S. Пополненная грамматика называется LR(k) для k>=0, если из условий

```
(1) S' =>* uAw => uvw,
(2) S' =>* zBx => 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->v - последнее правило, использованное в правом выводе цепочки uvw, то правило A->v должно применяться и в правом разборе при свертке uvy к uAy. Так как A дает v независимо от w, то LR(k) условие означает, что в

FIRST(w) содержится информация, достаточная для определения того, что uv за один шаг выводится из uA. Поэтому никогда не может возникнуть сомнений относительно того, как свернуть очередную правовыводимую цепочку пополненной грамматики. Кроме того, для LR(k) грамматики известно, когда допускается входная цепочка.

Основная разница между LL- и LR-грамматиками заключается в следующем. Чтобы грамматика была LR(k), необходимо распознавать вхождение правой части правила вывода, просмотрев все, что выведено из этой правой части и заглянув на k входных символов вперед. Это требование существенно менее строгое, чем требование для LL(k) грамматики, когда необходимо определить применимое правило, видя только первые k символов, выводимых из его правой части. Класс LL-грамматик является собственным подклассом LR. Рассмотрим теперь конструирование таблиц LR-анализатора.

LR(1) *ситуацией* называется пара [A->u.v,a], где A->uv - правило грамматики, а a - терминал или правый концевой маркер $. "1" указывает на длину второй компоненты ситуации, которая называется *аванцепочкой* ситуации. Аванцепочка не играет роли в ситуациях вида [A->u.v,a], где v не равно *e*, но ситуация вида [A->u.,a] ведет к свертке по правилу A->u только если следующим входным символом является a. Таким образом свертка по правилу A->u требуется только для тех входных символов a, для которых [A->u.,a] является LR(1) ситуацией в состоянии на верхушке магазина.

Будем говорить, что LR(1)-ситуация [A->u.v,a] *допустима для активного префикса* z, если существует вывод S=>*yAw=>yuvw, где z=yu и либо a - первый символ w, либо w равно *e* и a равно $ (рис. 3.18).

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

Пример 3.7. Рассмотрим грамматику

```
S -> BB
B -> aB | b
```

Существует правосторонний вывод S=>*aaBab=>aaaBab. Ситуация [B->a.B,a] допустима для активного префикса z=aaa, если в определении выше положить y=aa, A=B, w=ab, u=a, v=B. Существует также правосторонний вывод S*=>BaB=>BaaB. Из этого вывода видно, что для активного префикса Baa допустима ситуация [B->a.B,$].

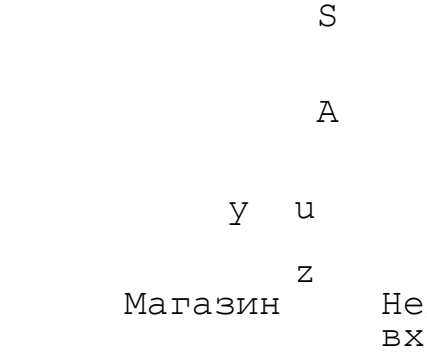
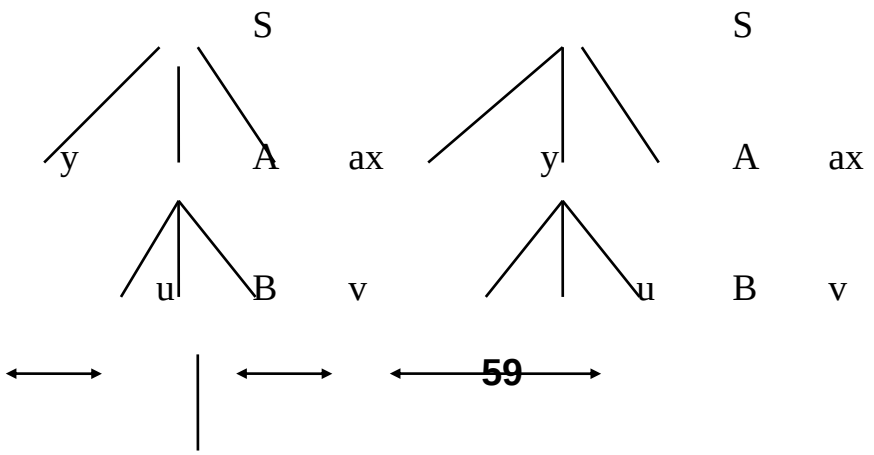


Рис. 3.18

Центральная идея LR-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, распознающие активные префиксы, а их группирование на самом деле есть процесс построения детерминированного конечного автомата из недетерминированного.

Для конструирования набора множеств допустимых LR(1)-ситуаций будут применяться пополненная грамматика G' и процедуры-функции closure и goto.

Рассмотрим ситуацию вида [A->u.Bv,a] из множества ситуаций, допустимых для некоторого активного префикса z. Тогда существует правосторонний вывод S=>*yAax=>yuBvax, где z=yu. Предположим, что из vax выводится терминальная строка bw. Тогда для некоторого правила вывода вида B->q имеется вывод S=>*zBbw=>zqbw. Таким образом [B->.q,b] также допустима для z и ситуация [A->uB.v,a] допустима для активного префикса zB. Здесь либо b может быть первым терминалом, выводимым из v, либо из v выводится *e* в выводе vax=>*bw и тогда b равно a. Т.е. b принадлежит FIRST(vax). Построение всех таких ситуаций для данного множества ситуаций, т.е. его замыкание, делает функция closure.



z bw zB

.q

Рис. 3.13

Алгоритм построения множеств LR(1)-ситуаций приведен ниже.

Алгоритм 3.8. Конструирование множеств LR(1)-ситуаций.

Алгоритм заключается в выполнении главной программы items, которая вызывает функции closure и goto.

```
SetOfItems closure(SetOfItems I)/*I - множество
ситуаций*/
{do
   {for (каждой ситуации [A->u.Bv,a] из I,
        каждого правила вывода B->w из G',
        каждого терминала b из FIRST(va),
         такого, что [B->.w,b] нет в I
       )
      добавить [B->.w,b] к I;
   }
   while (к I можно добавить новые ситуации);
      return I;
}
```

В анализаторах типа LR(0) при построении closure не учитываются терминалы из FIRST(va).

Если I - множество ситуаций, допустимых для некоторого активного префикса z, то goto(I,X) - множество ситуаций, допустимых для активного префикса zX.

```
SetOfItems goto(SetOfItems I,Symbol X)/*I - множество
ситуаций;
                      X - символ грамматики*/
{  Пусть [A->u.Xv,a] принадлежит I;
   Рассмотрим J - множество всех ситуаций вида
 {[A-uX.v,a]};
   return closure(J)
}
```

Работа алгоритма построения множества LR(1)-ситуаций начинается с того, что берется C - множество ситуаций {closure({[S'->.S,$]})}. Затем из имеющегося множества с помощью операции goto() строятся новые множества ситуаций. По-существу, goto(I,X) - переход конечного автомата из состояния I по символу X.

```
void items(Grammar G')
{  do
   {C={closure({[S'->.S,$]})};
    for (каждого множества ситуаций I из C,
       каждого символа грамматики X такого,
         что goto(I,X) не пусто и не принадлежит C
      )
      добавить goto(I,X) к C;
   }
   while (к C можно добавить множество ситуаций);
}
```

Пример 3.8. Рассмотрим пополненную грамматику примера 3.5.

```
    E'-> E
 1) E -> E + T
 2) E -> T
 3) T -> T * F
 4) T -> F
 5) F -> id
```

Множество ситуаций и переходов для этой грамматики приведены на рис. 3.19.

В каждый момент анализа в магазине находится активный префикс, который соответствует последовательности переходов из начального состояния (I_0) в текущее. Свертка - это замена суффикса префикса и переход в новое состояние, которое определяется из таблицы goto по символу левой части свернутого правила и предыдущему символу магазина, который соответствует состоянию после разбора префикса. Для определения этого нового состояния на верхушке хранится состояние, соответствующее активному префиксу без этого суффикса. Новое состояние определяется как goto[предыдущее состояние, новый нетерминал].

Рассмотрим теперь, как по множеству LR(1)-ситуаций строятся функции действия и переходов LR(1)-анализатора. Функции действия и переходов представляются таблицей.

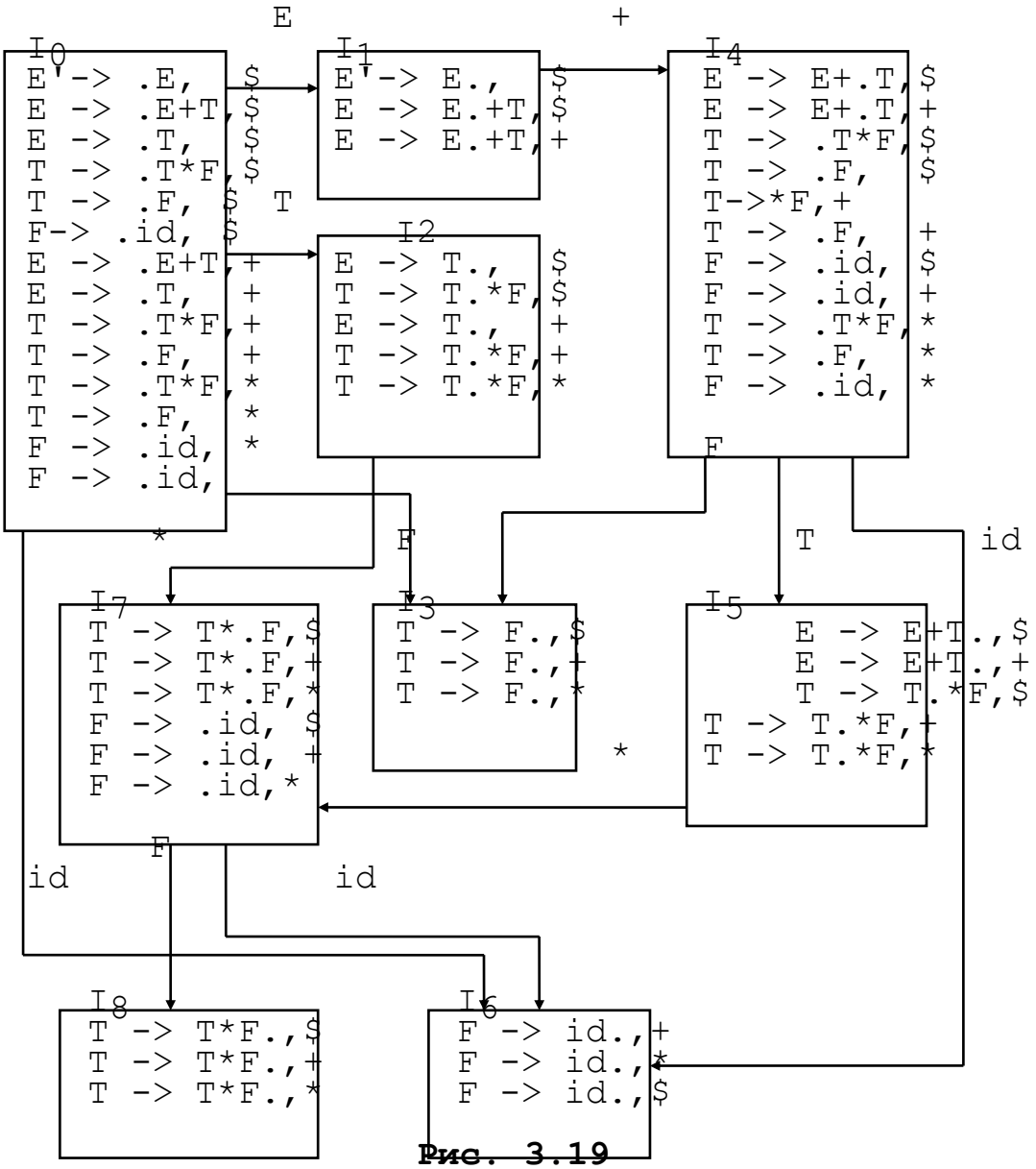


Рис. 3.19

Алгоритм 3.9. Построение канонических таблиц LR анализатора.

Шаг1. Строим набор множеств LR(1)- ситуаций C={I_0,I_1,...,I_n} для G'.

Шаг 2. Состояние i анализатора строится из I_i. Действия анализатора для состояния i определяются следующим образом:

а) если [A->u.av,b] принадлежит I_i и goto(I_i,a)=I_j, то полагаем action[i,a]="shift j". Здесь a - терминал;

б) если [A->u.,a] принадлежит I_i, A#S', то полагаем action[i,a]="reduce A->u";

в) если [S'->S.,$] принадлежит I_i, полагаем action[i,$]="accept".

Шаг 3. Переходы для состояния i определяются следующим образом: если goto(I_i,A)=I_j, то goto[i,A]=j (здесь A - нетерминал).

Шаг 4. Все входы, не определенные шагами 2 и 3, полагаем равными "error".

Шаг 5. Начальное состояние анализатора строится из множества, содержащего ситуацию [S'->.S,$].

Если при применении этих правил возникает конфликт, т.е. в одном и том же множестве может быть более одного варианта действий (либо сдвиг/свертка, либо свертка/свертка), говорят, что грамматика не является LR(1), и алгоритм завершается неуспешно.

Таблица, получающаяся из функций анализатора action и goto в результате работы Алгоритма 3.10, называется *канонической таблицей* LR(1)-анализатора. LR-анализатор, работающий по этой таблице, называется *каноническим* LR-анализатором. Если функция анализатора action не содержит неоднозначно определенных входов, то грамматика называется LR(1)-грамматикой.

3.3.4. Конфликты разбора типа сдвиг-свертка

Если грамматика не является LR(1), то анализатор типа сдвиг-свертка для нее может достигнуть конфигурации, в которой 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.

Пример 3.9. Рассмотрим вновь следующую грамматику оператора if-then-else:

```
St -> if Ex then St
      | if Ex then St else St
      | ...
```

Если анализатор типа сдвиг-свертка находится в конфигурации

```
Магазин                    Вход
... if Ex then St       else ... $
```

то нельзя определить, является ли if Ex then St 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 St -> if Ex then St или сдвиг else, а затем разбор другого St и завершение альтернативы if Ex then St else St. Таким образом нельзя сказать, нужно ли в этом случае делать сдвиг или свертку, так что грамматика не LR(1).

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

```
St -> CondSt | UnCondSt
CondSt -> IfThenSt | IfThenElseSt
FullSt -> IfThenElseSt | UnCondSt
IfThenElseSt -> if Ex then FullSt else St
IfThenSt ->if Ex then St
```

3.3.5. Восстановление после синтаксических ошибок

Одним из простейших методов 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. Тогда s - это, например, точка с запятой или end.

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.

4.1. Преобразователи с магазинной памятью

Преобразователем с магазинной памятью (МП-преобразователем) называется восьмерка P=<Q,T,Г,П,Ф,q_0,Z_0,F>, где Q - множество состояний, T - конечный входной алфавит, Г - конечный алфавит магазинных символов, П - конечный выходной алфавит, Ф - отображение множества Qx(T U {*e*})xГ в множество конечных подмножеств множества QxГ*xП*, q_0<-Q - начальное состояние, Z_0<-Г - начальный символ магазина, F<=Q - множество заключительных состояний.

Определим *конфигурацию* преобразователя P как четверку <q,x,u,y>, где q<-Q - состояние, x<-T* - цепочка на входной ленте, u<-Г* - содержимое магазина, y<-П* - цепочка на выходной ленте, выданная вплоть до настоящего момента. Если Ф(q,a,Z) содержит (r,u,z), то будем писать <q,ax,Zw,y>|-<r,x,uw,yz> для любых x<-A*, w<-Г* и y<-П*. Рефлексивное и транзитивное замыкание отношения |- будем обозначать |-*.

Цепочку y назовем *выходом* для x, если <q_0,x,Z_0,*e*>|-*<q,*e*,u,y> для некоторых q<-F и u<-Г*. *Переводом* (или *преобразованием*), определяемым МП-преобразователем P (обозначается t(P)), назовем множество {(x,y)|<q_0,x,Z_0,*e*>|-*<q,*e*,u,y> для некоторых q<-F и u<-Г*}. Будем говорить, что МП-преобразователь P=<Q,T,Г,П,Ф,q_0,Z_0,F> *детерминированный* (ДМП-преобразователь), если выполняются следующие условия:

1) для всех q<-Q, a<-T U {*e*} и Z<-Г множество Ф(q,a,Z) содержит не более одного элемента,
2) если Ф(q,*e*,Z)#{}, то Ф(q,a,Z)={} для всех a<-T.

Пример 4.1. Перевод арифметического выражения в ПОЛИЗ.

ПОЛИЗ - Польская инверсная запись или, что то же, постфиксная запись арифметических выражений. Трансляция может определяться следующим ДМП:

```
Q={q0,+,*,),$};
T={буквы,+,*,(,),$}, здесь $ - концевой маркер;
Г={Z0,(,+,*};
П={буквы,+,*};
```

Функция переходов определяется таблицей на рис. 4.1.

Г	Q	T
Z_0 Z_0 Z_0	q_0 q_0 q_0	буква (проч
(+,*	))	
+ * проч	* + +,*	
+,* Z_0	$ $	

Г*	Q	п
z_0 z_0(z_0	q_0 q_0 проч	буква
e e	q_0)	+,*
++ *+ проч	q_0 q_0 {+,*}q_0	
e e	$	+,*

Рис. 4.1

Стек	Состояние	Вход	Выход
Z_0	q_0	a*(b+c)$	a
Z_0	q_0	*(b+c)$	
Z_0	*	(b+c)$	
Z_0*	q_0	(b+c)$	
Z_0*(	q_0	b+c)$	b
Z_0*(	q_0	+c)$	
Z_0*(	+	c)$	
Z_0*(+	q_0	c)$	c
Z_0*(+	q_0	)$	
Z_0*(+	)	$	+
Z_0*(	)	$	
Z_0*	q_0	$	
Z_0*	$		*
Z_0	$		

Рис. 4.2

Последовательность состояний автомата и магазина на строке a*(b+c) изображены в таблице рис. 4.2.

4.2. Синтаксически управляемый перевод

Схемой синтаксически управляемого перевода (или трансляции, сокращенно: СУ-схемой) называется пятерка Tr=(N,T,П,R,S), где

N - конечное множество нетерминальных символов;

T - конечный входной алфавит;

П - конечный выходной алфавит;

R - конечное множество правил перевода вида A->u, $A_1=v_1$, ... , $A_m=v_m$, удовлетворяющих следующим условиям:

- каждый символ, входящий в v_1, ..., v_m, либо принадлежит П, либо является B_k для B<-N и B входит в u,
- если u имеет более одного вхождения символа B, то каждый символ B_k во всех v соотнесен (верхним индексом) с конкретным вхождением B;

S - начальный символ, выделенный нетерминал из N.

A->u называют *входным правилом* вывода, A_i - *переводом* нетерминала A, $A_i=v_i$ - *элементом* перевода, связанным с этим правилом перевода. Если через P обозначить множество входных правил вывода всех правил перевода, то G=(N,T,P,S) будет *входной грамматикой* для Tr. Если в СУ-схеме Tr нет двух правил перевода с одинаковым входным правилом вывода, то ее называют *семантически однозначной*. *Выход* СУ-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 t(Tr), *определяемым СУ-схемой* Tr, назовем множество {(x,y)|x имеет дерево разбора во входной грамматике для Tr и y - значение выделенного символа перевода S в корне этого дерева}. Если Tr=<N,T,П,R,S> - СУ-схема, то т(Tr) называется *синтаксически управляемым переводом* (СУ-переводом).

Пример 4.2. Рассмотрим формальное дифференцирование выражений, включающих константы 0 и 1, переменную x и функции sin, cos, + и *. Такие выражения порождает грамматика

```
E -> E+T | T
T -> T*F | F
F -> (E) | sin(E) | cos(E) | x | 0 | 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 E2. Законы дифференцирования таковы:

```
d(f(x)+g(x))=df(x)+dg(x)                                    dx=1
 d(f(x)*g(x))=f(x)*dg(x)+g(x)*df(x)                          d0=0
 dsin(f(x))=cos(f(x))*df(x)                                  d1=0
 dcos(f(x))=-sin(f(x))df(x)
```

Эти законы реализуются СУ-схемой:

```
E -> E+TE1=E1+T1    F -> sin(E)    F1=sin(E1)
        E2=E2+T2                   F2=cos(E1)*(E2)
 E -> T  E1=T1      F -> cos(E)    F1=cos(E1)
        E2=T2                      F2=-sin(E1)*(E2)
 T -> T*F   T1=T1*F1    F -> x     F1=x
            T2=T1*F2+T2*F1         F2=1
 T -> F     T1=F1       F -> 0     F1=0
            T2=F2                  F2=0
 F -> ( E ) F1=(E1)     F -> 1     F1=1
            F2=(E2)                F2=0
```

Дерево вывода для sin(cos(x))+x приведено на рис. 4.3.

Теорема 4.1. Если число вхождений каждого нетерминала в слове v не превосходит 1, то t(Tr) является КС-языком. Обратное не всегда верно [5].

Пример 4.2. T=<{S,A},{a},{a,b},{S->A,AbAbA;A->a,a;A->aA,aA}>. Здесь входной язык $\{a^n|n>=1\}$, выходной $\{a^nba^nba^n\}$. Выходной язык не КС.

Теорема 4.2. Для каждого магазинного преобразователя существует эквивалентная СУ-схема [4].

Обратное, вообще говоря, не верно.

Определение. Семантически однозначная СУ-схема Tr=(N,T,П,R,S) называется *простой*, если для каждого правила A->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).

Теорема 4.3. Пусть Tr=(N,T,П,R,S) - простая СУ-схема. Существует такой МП-преобразователь P, что t(P)=t(Tr) [5].

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

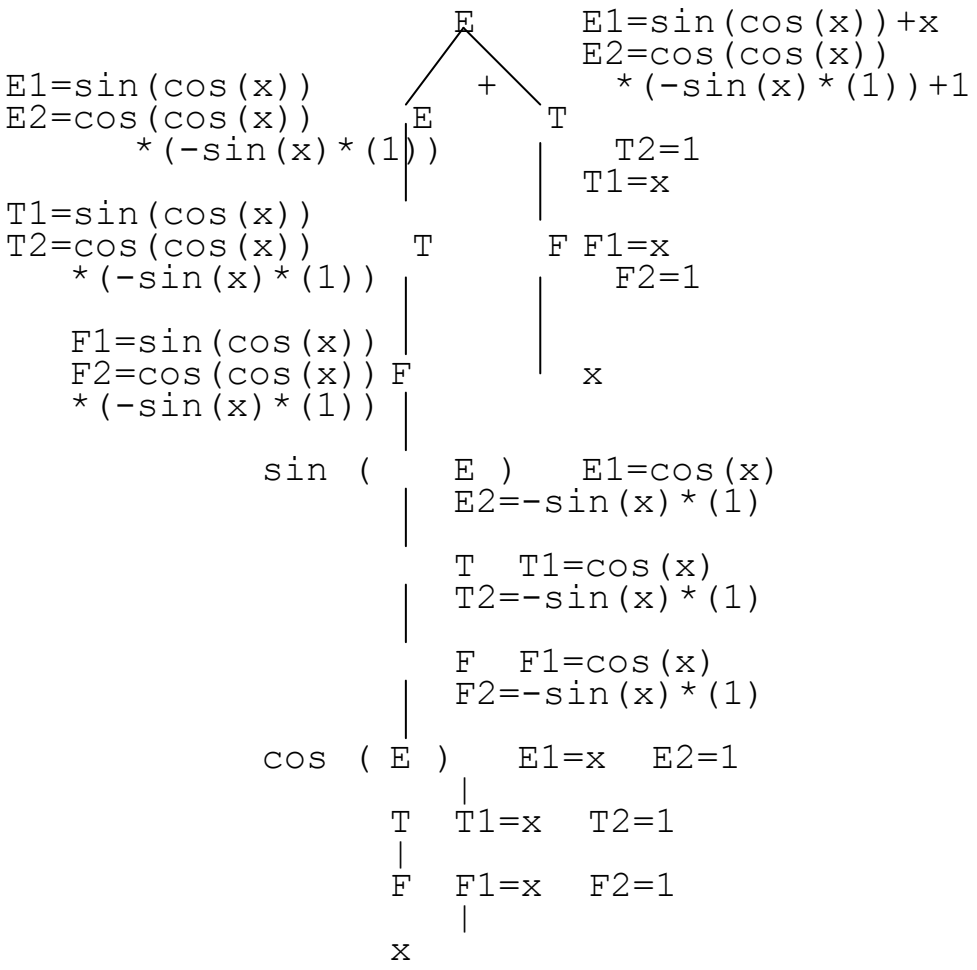


Рис. 4.3

Теорема 4.4. Пусть Tr=(N,T,П,R,S) - семантически однозначная простая СУ-схема, входной грамматикой которой служит LL(k)-грамматика. Тогда перевод {x$,y)|(x,y)<-t(Tr)} можно осуществить детерминированным МП-преобразователем [5].

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

Пример 4.3. Рассмотрим простую СУ-схему T с правилами

S -> Sa, aSa
S -> Sb, bSb
S -> *e*, *e*

Входная грамматика является LR(1) грамматикой, но не существует ДМП-преобразователя, определяющего перевод {(x$,y)|(x,y)<-t(Tr)} [5].

Определение. Назовем СУ-схему Tr=<N,T,П,R,S> *постфиксной*, если каждое правило из R имеет вид A->u,v, где v<-N*П*.

Иными словами, каждый элемент перевода представляет собой цепочку из нетерминалов, за которыми следует цепочка выходных символов.

Теорема 4.5. Пусть Tr=<N,T,П,R,S> - семантически однозначная простая постфиксная СУ-схема, входной грамматикой которой служит LR(k)-грамматика. Тогда перевод {(x$,y)|(x,y)<-t(Tr)} можно осуществить детерминированным МП-преобразователем [5].

4.3. Атрибутные грамматики

Среди всех формальных методов описания языков программирования атрибутные грамматики 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.

4.3.1. Определение атрибутных грамматик

Пусть G - КС-грамматика: G=<T,N,P,Z>, где T, N, P, Z, - соответственно, множество терминальных символов, нетерминальных символов, множество правил вывода и аксиома грамматики. Правила вывода КС-грамматики будем записывать в виде

$p: X_0 \to X_1 \ldots X_n\langle p\rangle$

и будем предполагать, что G - редуцированная КС-грамматика, т.е. в ней нет нетерминальных символов, для которых не существует полного дерева вывода , в которое входит этот нетерминал.

С каждым символом X <- N **U** T свяжем множество A(X) *атрибутов* символа X. Некоторые из множеств A(x) могут быть пусты. Запись a(X) означает, что a <- A(X).

С каждым правилом вывода p <- P свяжем множество F *семантических правил*, имеющих следующую форму:

$a_0\langle i_0\rangle = f^p_{a0\langle i0\rangle}(a_1\langle i_1\rangle, \ldots , a_j\langle i_j\rangle),$

где i_k <- $[0,n_p]$ - номер символа правила p, а $a_k\langle i_k\rangle$ - атрибут символа X_{ik} , т.е. $a_k\langle i_k\rangle$ <- $A(X_{ik})$.

В таком случае будем говорить , что $a_0\langle i_0\rangle$ "зависит" от

$a_1<i_1>,...,a_j<i_j>$ или что $a_0<i_0>$ "вычисляется по" $a_1<i_1>,...,a_j<i_j>$. В частном случае j может быть равно нулю, тогда будем говорить, что атрибут $a_0<i_0>$ "получает в качестве значения константу".

КС-грамматику, каждому символу которой сопоставлено множество атрибутов, а каждому правилу вывода - множество семантических правил, будем называть *атрибутной грамматикой* (AG).

Назовем атрибут $a(X_0)$ *синтезируемым*, если одному из правил вывода p: $X_0 \to X_1 \ ... \ X_{np}$ сопоставлено семантическое правило $a<0>=f_{a<0>}(...)$. Назовем атрибут $a(X_i)$ *наследуемым*, если одному из правил вывода $p:X_0 \to X_1 \ ... \ X_i \ ... \ X_{np}$ сопоставлено семантическое правило $a<i>=f_{a<i>}(...)$, $i \leftarrow [1,n_p]$. Множество синтезируемых атрибутов символа X обозначим через S(X), наследуемых атрибутов - через I(X).

Будем считать, что значение атрибутов терминальных символов - константы, т.е. их значения определены, но для них нет семантических правил, определеяющих их значения.

В качестве примера рассмотрим атрибутную грамматику, вычисляющую значение дробного числа, представленного в десятичной форме.

```
Число ::= Целая_часть '.' Дробная_часть=>
   Значение<0>=Значение<1>+Значение<3>;
   Порядок<3>=1.
Целая_часть ::= e=>
   Значение<0>=0; Порядок<0>=0.
Целая_часть ::= Цифра Целая_часть =>
   Значение<0>=Значение<1>*10**Порядок<2>+Значение<2>;
   Порядок<0>=Порядок<2>+1.
Дробная_часть ::= e =>
   Значение<0>=0.
Дробная_часть ::= Цифра Дробная_часть=>
   Значение<0>=Значение<1>*10**(-Порядок<0>)
              +Значение<2>;
   Порядок<2>=Порядок<0>+1.
```

4.3.2. Атрибутированное дерево разбора

Если дана атрибутная грамматика AG и цепочка, принадлежащая языку, определяемому G, то можно построить дерево разбора этой цепочки в грамматике G. В этом дереве каждая вершина помечена символом грамматики G. Припишем теперь каждой вершине множество атрибутов, сопоставленных символу, которым помечена эта вершина. Атрибуты,

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

Между вхождениями атрибутов в дерево разбора существуют зависимости, определяемые семантическими правилами, соответствующими примененным синтаксическим правилам.

4.3.3. 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 автоматизации 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, генерированных 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, в котором предполагается возможность управления порядком исполнения операторов. Этот порядок привязывается к обходу атрибутированного

дерева разбора сверху вниз слева направо. Все атрибуты должны быть вычислены за один такой обход. Атрибутные грамматики, обладающие таким свойством, называются L-атрибутными.

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.

```
Атрибутная грамматика ::= 'ALPHABET'
          ( ОписаниеНетерминала ) ( Правило )
ОписаниеНетерминала ::= ИмяНетерминала
                  '::' [( ОписаниеАтрибутов / ';')] '.'
ОписаниеАтрибутов ::= ( ИмяАтрибута / ',') ':' Тип
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 переменных). 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 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 (рис. 4.4).

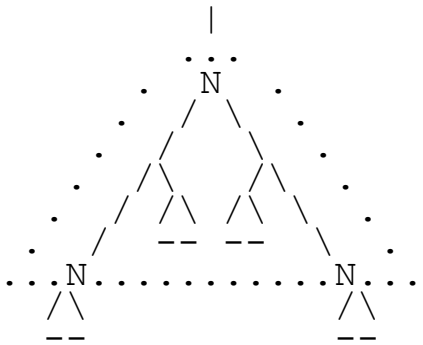


Рис. 4.4

Значение терминального символа доступно через атрибут VAL соответствующего типа.

4.3.4. Классы атрибутных грамматик и их реализация

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

Простейшими такими подклассами атрибутных грамматик, вычисления всех атрибутов для которых может быть осуществлено одновременно с синтаксическим анализом, являются S-атрибутные и L-атрибутные грамматики. Атрибутная грамматика называется S-атрибутной, если все атрибуты на любом дереве для этой грамматики могут быть вычислены снизу-вверх слева-направо (т.е. атрибутная грамматика имеет только синтезированные атрибуты). Все такие атрибуты могут быть вычислены в процессе разбора снизу-вверх.. Грамматика называется L-атрибутной, если все атрибуты на любом дереве для этой грамматики могут быть вычислены за один обход дерева сверху-вниз слева-направо. Все такие атрибуты могут быть вычислены в процессе разбора сверху-вниз.

Вычисление атрибутов для L-атрибутных грамматик можно производить в процессе рекурсивного спуска. Для этого в каждой функции, соответствующей нетерминалу, надо определить формальные параметры, передаваемые по значению, для наследуюмых атрибутов, и формальные параметры, передаваемые по ссылке, для синтезируемых атрибутов. В качестве примера рассмотрим реализацию приведенной выше атрибутной грамматики.

```
void int_part(float * V0, int * P0)
{if (Map[InSym]==Digit)
  {  int I=InSym;
     InSym=getInSym();
     float V2;
     int P2;
     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;
    InSym=getInSym();
    float V2;
    int P2=P0+1;
    fract_part(&V2,P2);
    V0=I*exp(-P2*ln(10))+V2;
    P0=P2+1;
 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;
}
```

Глава 5. Контекстные условия языков программирования

5.1. Описание областей видимости и блочной структуры

Задачей контекстного анализа является установление правильности использования объектов. Наиболее часто решаемой задачей является определение существования объекта и соответствия его использования контексту, что осуществляется с помощью анализа типа объекта.

Таким образом необходимо хранить объекты, 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

$E=\{DS_1,...DS_n\}$.

Каждая компонента - это множество объявлений, представляющих собой пары <имя,тип>:

$DS_i=\{$<имя,тип>$\}$,

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

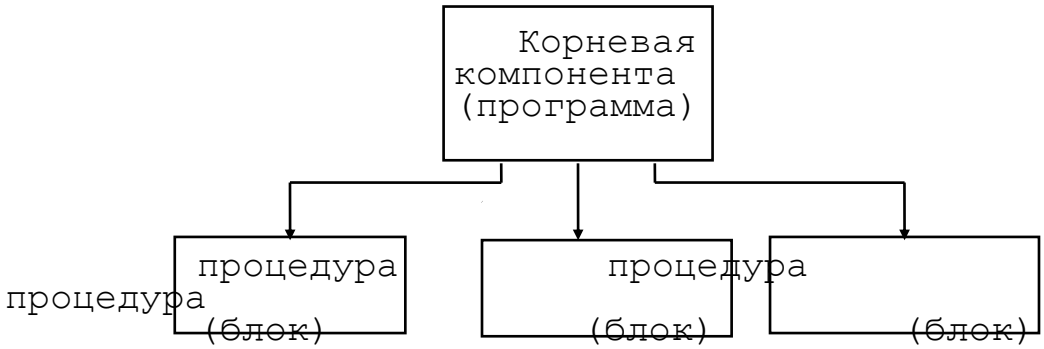


Рис. 5.1

Между компонентами DS_i и DS_j имеет место отношение "DS_i включает DS_j" тогда и только тогда, когда любой объект из DS_i может

быть доступен из DS_j (конкретный способ доступа определяется правилами видимости языка), но не наоборот. Компоненты образуют дерево. Это дерево соответствует блокам или процедурам (рис. 5.1)

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.

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" компоненты.

Для обозначения участков программы, в которых доступны те или иные описания, используются понятия "*область действия*" и "*область 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.

В разных языках понятия области действия и области видимости уточняются по-разному. В дальнейшем изложение ведется на примере языка Модула -2.

5.2. Структура среды Модулы-2

Имеются четыре рода языковых конструкций, которые могут содержать описания: 1) программный модуль или модуль реализации; 2) модуль определения; 3) процедура; 4) локальный модуль.

"Корневую" компоненту среды в Модуле-2 образует программный модуль или модуль реализации. Объекты этой компоненты могут быть описаны в самом модуле или могут быть импортированы из других модулей определений. Такой импорт может быть квалифицированным (from M import X,Y, ...;) или не квалифицированным (import M;).

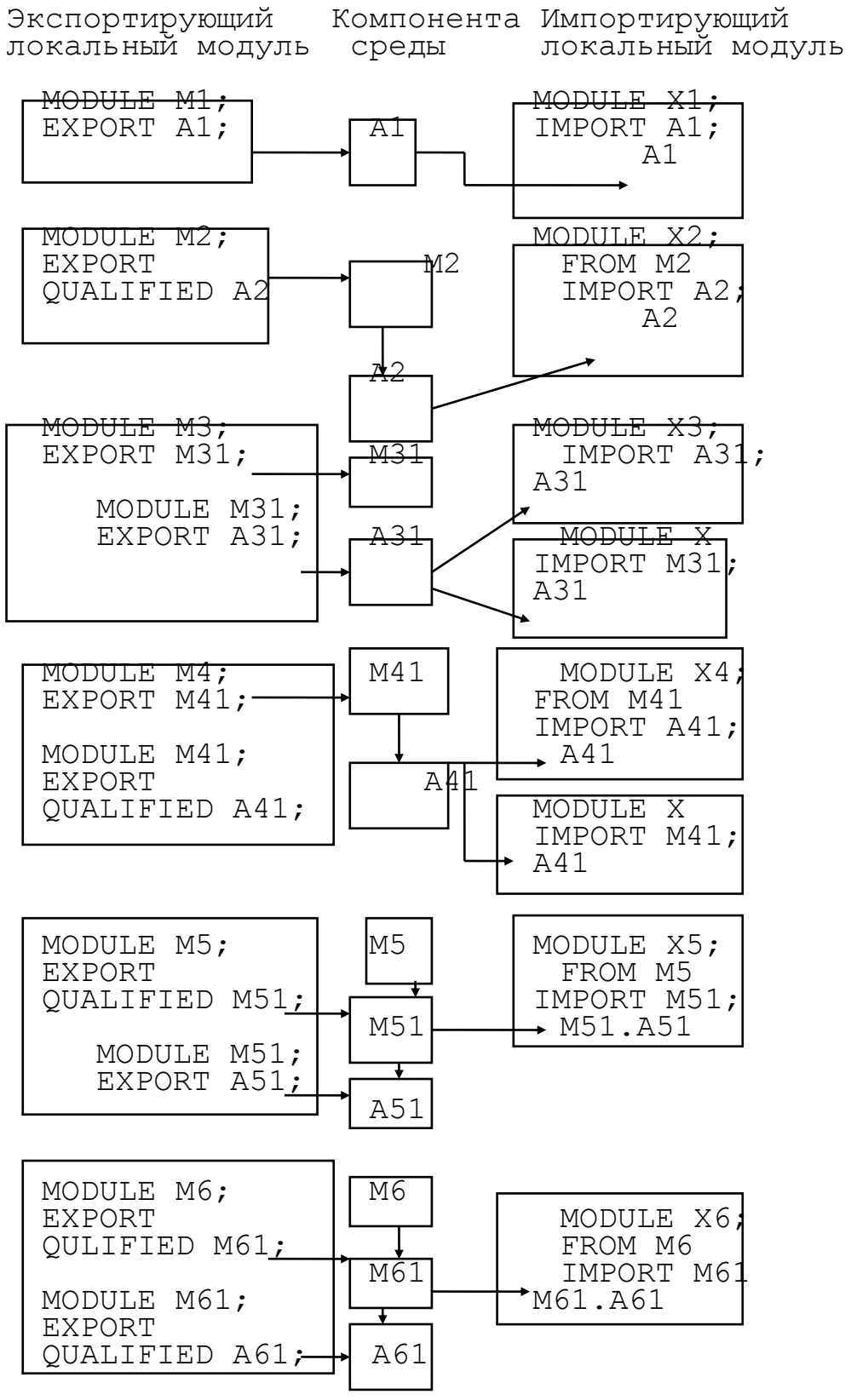


Рис. 5.2

В первом случае импортированные объекты становятся элементами среды

данного модуля, во втором - сам импортированный модуль становится элементом среды, а его объекты могут быть доступны через указание имени модуля (M.X).

Область действия объектов, описанных в локальном модуле, может состоять из самого этого локального модуля или из охватывающего его блока, если объект экспортируется из локального модуля. Схему импорта в локальный модуль можно пояснить рис. 5.2. Существует предопределенная компонента, объекты которой доступны во всех других компонентах (если они там не переопределены). Эта компонента включает в себя типы данных такие как, integer, real, boolean, char, word, address, proc, константы true, false, nil, процедуры adr, tsize, cap, small, chr, inc, dec, float, halt, hihg, odd, ord, trunc, val, excl, incl, max, min, size, abs.

Элементом описания может быть процедура или локальный модуль, имеющие свой список описаний. Процедура образует новую компоненту среды, а локальный модуль - нет (рис. 5.3).

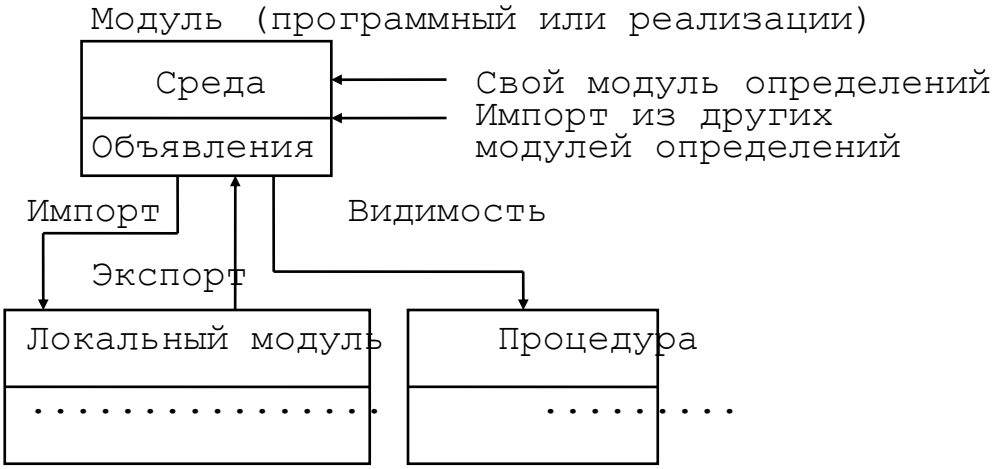


Рис. 5.3

Объекты локального модуля являются объектами охватывающей компоненты, но с ограниченными областями видимости. Внутри локального модуля доступны те и только те объекты этой компоненты, которые явно импортированы в локальный модуль. И наоборот: объекты локального модуля могут быть экспортированы в охватывающую компоненту. В то же время объекты процедуры образуют новую компоненту, поскольку объекты этой компоненты могут быть доступны в процедуре. Конфликт имен при этом не противоречит определению компоненты: объект охватывающей компоненты может быть виден (если внутри данной процедуры не описан объект с тем же именем).

Среда состоит из отдельных объектов, реализуемых как записи. Состав полей записи вообще говоря зависит от 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.

5.3. Занесение в среду и поиск объектов

Поскольку блоки образуют иерархию на дереве разбора программы, при поиске объекта мы можем с помощью атрибута типа "указатель на блок" переходить от блока к охватывающему блоку. Если теперь у каждого блока есть атрибут, указывающий, является ли блок процедурой или модулем, то легко реализовать сочетание блочной структуры со средствами управления видимостью. Кроме того, корень дерева имеет атрибут, соответствующий предопределенной компоненте, так что через этот глобальный атрибут доступны все предопределенные описания (рис. 5.4).

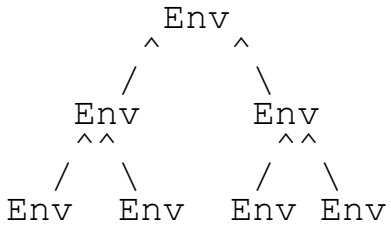


Рис. 5.4

В качестве типов данных атрибутов мы будем использовать множество. 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 локализована в теле оператора и 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. Каждый локальный модуль имеет атрибут - множество импортируемых объектов и атрибут - множество экспортируемых объектов. Экспортируемые модулем объекты в соответствии с правилами экспорта включаются в компоненту среды, в которую входит сам модуль. Ниже приведен фрагмент описания контекстных условий языка Модула-2 (в фигурные скобки заключены комментарии).

```
ALPHABET
Prog:: KEY TName SETOF Element Env.
```

Предопределенная компонента. Идентификаторы имеют тип Name.

```
Block :: KEY Name SETOF Element Env;
         bool Kind;
         Block * Pred.
```

Арибут Kind равен true для процедур и false для модулей. Env - множество объявлений блока. Pred - указатель на охватывающий блок.

```
Mod_Head :: Element * Entry;
            SETOF Import_Pair Imp_Set;
            bool Mode.
```

Imp_Set - множество импортируемых модулем объектов; тип Import_Pair - запись из двух полей: Imp_Name:Name - имя импортируемого объекта, и Name_Set:Set of Name - список импортируемых из модуля объектов, если импорт квалифицированный, и Nil, если неквалифицированный; Entry - указатель на вход для модуля; Mode - признак квалифицированного экспорта.

```
Import :: SETOF Name Imp_Set.
```

Imp_Set - список квалифицированного импорта.

```
Export :: bool Mode.
```

Mode - признак квалифицированного экспорта.

```
From :: bool Qual; TName Name.

Qual :: bool Qual.
Ident_List :: SETOF Name Ident_Set.
Type_Des :: Element * Exit.
```

Qual - признак квалифицированного импорта; Name - имя модуля, из которого делается импорт; Ident_Set - список идентификаторов; Exit - указатель на описание типа.

```
RULE
Declaration ::= 'procedure' Ident [ Formal_Param ] ';'
                Block ';'
SEMANTICS
Element* Entry;
Kind<5>=true;
2:if (Find(Val<2>,Env<Block>)!=NULL)
     Error("Identifire declared twice");
  Entry=Include(Val<2>,Env<Block>);
  Entry->Object=ProcObject.
```

Помещение процедуры; ищем объект с данным именем в текущей компоненте; Entry - указатель на вход для процедуры. Если объект с таким именем уже есть, - ошибка.

```
RULE
Declaration ::= 'module' Ident Mod_Head Block ';'
SEMANTICS
Import_Pair M;
Element * V;
if (Find(Val<2>,Env<Block>)!=NULL)
   Error("Identifire declared twice");

Entry<3>=Include(Val<2>,Env<Block>);
Entry<3>->Object=LocalModuleObject;
Kind<4>=false;
3: for (M in Imp_Set<3>) Inc_Imp(M,Env<4>);
if (Mode<3>==NotQual)
 for (V in Entry<3>->Exp_Set) Export(V,Env<Block>).
```

Помещение модуля; ищем объект с данным именем в текущей компоненте.

Если такой уже есть, - ошибка. Заносим в текущую компоненту среды объект-модуль. Множество Imp_Set - это множество импортируемых модулем объектов. Элементы этого множества - пары, первая компонента которых - имя импортируемого модуля, вторая - список импортируемых из него объектов. Если вторая компонента Nil, то импорт неквалифицированный. Если импортируемый объект M - модуль и если M импортируется неквалифицированно (IMPORT M), то процедура Inc_Imp включает M в среду текущего модуля; если импорт квалифицированный (FROM M IMPORT ...), то M имеет список импорта и процедура Inc_Imp включает в текущую компоненту те экспортируемые из M объекты, которые перечислены в списке квалифицированного импорта; если при этом импортируемый объект - в свою очередь модуль, то из M рекурсивно импортируется все его экспортные объекты.

Атрибут Mode вычисляется в правиле для экспорта и определяет, осуществляется ли квалифицированный экспорт (EXPORT QUALIFIED ...) или нет (EXPORT ...). Процедура Export в случае неквалифицированного экспорта EXPORT A1,A2,... все объекты экспортного списка модуля заносит в компоненту среды, охватывающую модуль; если Ai - модуль, его экспорт обрабатывается рекурсивно; если экспорт квалифицированный, то в охватывающую модуль компоненту попадает только сам модуль со списком экспорта.

```
RULE
Block ::= [( Declaration )] [ Block_St_Seq ] 'end'
Ident
SEMANTICS
0:Init(Env<0>);
  Pred<0>=&<Block>.
```

Атрибут Pred - указатель на охватывающий блок.

```
RULE
Mod_Head ::= [ Priority ] ';' [ ( Import ) ] [ Export
]
SEMANTICS
4E: Mode<4>=NotQual;
Entry<0>->Mode=Mode<4>;
Mode<0>=Mode<4>;
0:Init(Imp_Set<0>).
```

Инициализируется список импорта. Тип экспорта заносится в описание модуля.

```
RULE
Import ::= [ From ] 'import' ( Ident /',' ) ';'
```

```
SEMANTICS
Import_Pair Tmp;
0:Init(Imp_Set<0>);
1E: Qual<1>=false;
3A:if (Qual<1>) Include(Val<3>,Imp_Set<0>);
   else {Tmp.Name_Set=NULL;
         Tmp.Imp_Name=Name<3>;
         Include(Tmp,Imp_Set<Mod_Head>);
        }
if (Qual<1>)
       {Tmp.Name_Set=Imp_Set<0>;
        Tmp.Imp_Name=Name<1>;
        Include(Tmp,Imp_Set<Mod_Head>);
       }.
```

Если импорт квалифицированный, то Name<1> - имя модуля, из которого делается импорт. В этом случае Imp_Set<0> - множество импортируемых из него объектов. Идентификатор включается в список импорта из модуля, иначе идентификатор - имя модуля и он включается в список импортируемых модулей. Если импорт квалифицированный, включить в список импорта модуля весь список импортируемых объектов.

```
RULE
From ::= 'from'  Ident
SEMANTICS
Qual<0>=true;
Name<0>=Val<2>.
```

```
RULE
Export ::= 'export'  [ Qual ]  ( Ident /',' )
SEMANTICS
0: Init(Entry<Mod_Head>->Exp_Set);
2E: Mode<2>=NotQual;
    Mode<0>=Mode<2>;
3A: Include(Val<3>,Entry<Mod_Head>->Exp_Set).
```

Включить идентификатор в экспортный список модуля; множество экспортируемых модулем объектов заносится в поле Exp_Set : Key Name Set Of Element в таблице объектов, поскольку при использовании квалифицированного импорта или обращении M.V, где M - имя модуля, а V - имя экспортируемого объекта, необходимо иметь доступ к списку квалифицированного экспорта.

```
RULE
Qual ::= 'qualified'
```

```
SEMANTICS
Qual<0>=Qualified.

RULE
Declaration ::= 'var' ( Var_Decl ).

RULE
Var_Decl ::= Ident_List ':' Type_Des ';'
SEMANTICS
Element V;
for (V in Ident_Set<1>)
  {if (Find(V,Env<Block>)!=NULL)
      Error("Identifire declared twice");
   V.Object=VarObject;
   V.Type=Exit<3>;
   Include(V,Env<Block>)
   }.
```

V - рабочая переменная для элементов списка. Для каждого идентификатора из множества 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
Type_Des ::= Ident
SEMANTICS
Block * P;
P=&<Block>;
do{
  Exit<0>=Find(Val<1>,P->Env);
  if (P->Kind) P=P->Pred;
  }
while (Exit<0>==NULL)&&(P->Kind);
if (Exit<0>==NULL)
   Exit<0>=Find(Val<1>,Env<Prog>);
if (Exit<0>==NULL) Error("Type Description not
found");
else if (Exit<0>->Object!=TypeObject)
         {Error("Not type object");
          Exit<0>:=Nil;
```

```
}.
```

Рабочая переменная P - указатель на ближайший охватывающий блок. Переходя от блока от блоку, ищем объект в его среде. Если не нашли, то, если блок - процедура, перейти к охватывающей. Если дошли до границы модуля, попытаться найти объект в предопределенной компоненте. Если объект нашли, надо убедиться, что он - тип.

Зависимости атрибутов на дереве разбора приведены на рис. 5.5.

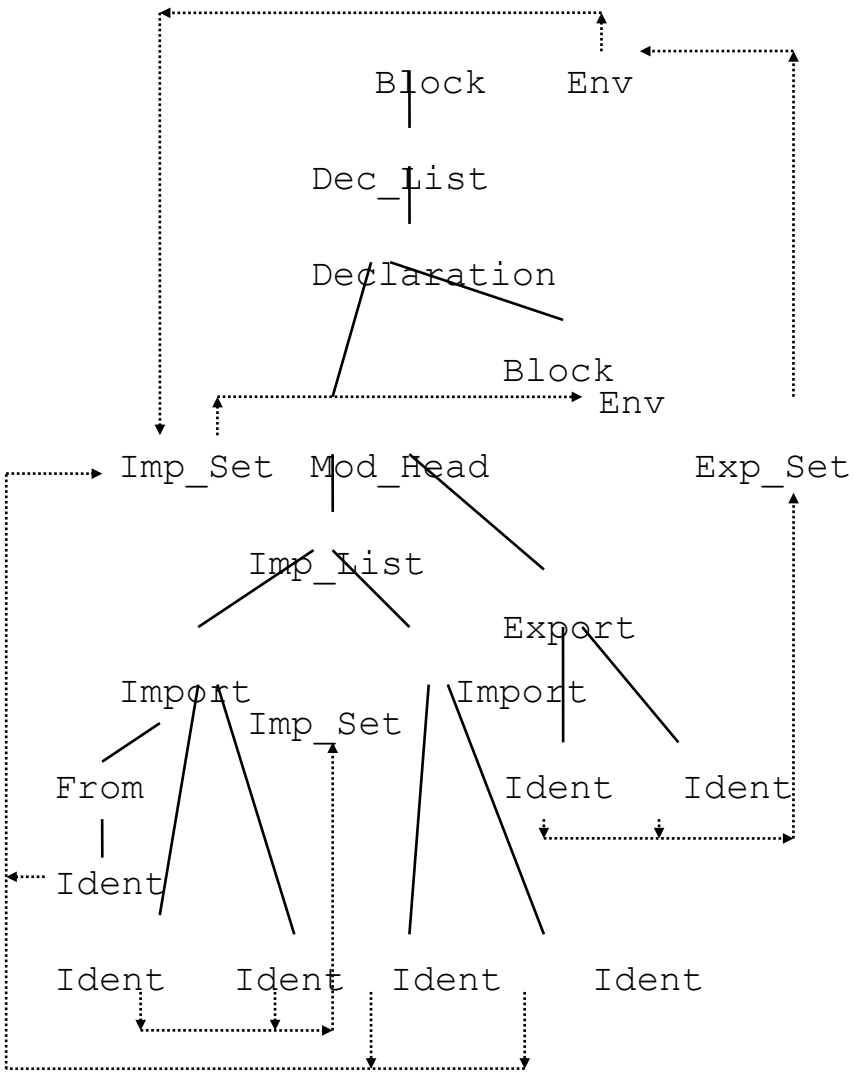


Рис. 5.5

Глава 6. Организация таблиц символов компилятора

В процессе работы компилятор хранит информацию об объектах программы. Как правило, информация о каждом объекте состоит из двух основных элементов: имени объекта и его свойств. Информация об объектах программы должна быть организована таким образом, чтобы поиск ее был по возможности быстрей, а требуемая память по возможности меньше. Кроме того, со стороны языка программирования могут быть дополнительные требования. Имена могут иметь определенную область видимости. Например, поле записи должно быть уникально в пределах структуры (или уровня структуры), но может совпадать с именем объектов 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 по 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 информации в компиляторах: таблицы идентификаторов, таблицы символов, способы реализации блочной структуры.

6.1. Таблицы идентификаторов и таблицы символов

Как уже было сказано, информацию об объекте обычно можно разделить на две части: имя (идентификатор) и описание. Удобно эти характеристики объекта хранить по отдельности. Это обусловлено двумя причинами: 1) символьное представление идентификатора может иметь неопределенную длину и быть довольно длинным; 2) как уже было сказано, различные объекты в одной области видимости и/или в разных могут иметь одинаковые имена и незачем занимать память для повторного хранения идентификатора. Таблицу для хранения идентификаторов называют *таблицей идентификаторов*, а таблицу для хранения свойств объектов - *таблицей символов*. В таком случае одним из свойств объекта становится его имя и в таблице символов хранится указатель на

соответствующий вход в таблицу идентификаторов.

6.2. Таблицы идентификаторов

Если длина идентификатора ограничена (или имя идентифицируется по ограниченному числу первых символов идентификатора), то таблица идентификаторов может быть организована в виде простого массива строк фиксированной длины, как это изображено на рис. 6.1. Некоторые входы могут быть заняты, некоторые - свободны.

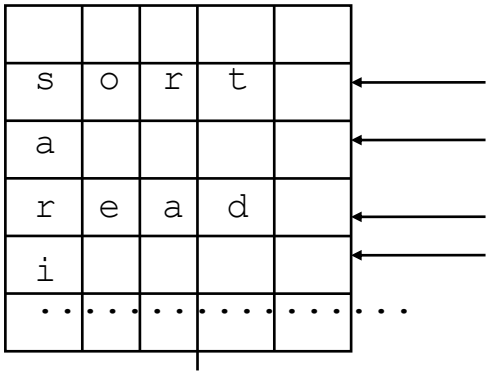


Рис. 6.1

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. Поиск в такой таблице может быть организован методом повторной расстановки. Суть его заключается в следующем.

Пусть число элементов массива равно N. Определим некоторую функцию h (первичную функцию расстановки), определенную на множестве идентификаторов и принимающую значения 0<=h(id)<=N-1, id - идентификатор. Если элемент таблицы h(id) свободен, 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(id). Если они равны - идентификатор нашли, если нет - надо продолжать поиск дальше. Для этого вычисляют вторичную функцию расстановки h2(h). Вновь возможны четыре варианта: либо элемент таблицы свободен, либо нашли идентификатор, либо таблица вся просмотрена и идентификатора нет,

либо надо продолжать поиск. Для продолжения поиска применяют вновь функцию h3(h2), и т.д. Как правило, hi=h2 для i>=2. Аргументом функции h2 является целое в диапазоне [0,N-1] и она может быть быть устроена по-разному. Приведем три варианта.

1) h2(i)=(i+1) mod 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) h2(i)=(i+k) mod N, где k и N взаимно просты.

По-существу это предыдущий вариант, но элементы накапливаются не в последовательных элементах, а "разносятся".

3) h2(i)=(a*i+c) mod 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 [6].

```
void Search(String id, boolean * Yes, int * Point)
{int H0=h(id),H=H0;
  while (1)
   {if (!IdComp(H,id))
      {*Yes=true;
       *Point=H;
       return;
      }
    else if (Empty(H))
      {*Yes=false;
        *Point=H;
        return;
      }
    else H=h2(H);
    if (H==H0)
      {*Yes=false;
      *Point=NULL;
    return;
}  }  }
```

Поиск в таблице можно описать функцией, приведенной выше. Функция IdComp(H,id) сравнивает элемент таблицы по входу H с идентификатором и вырабатывает 0, если они равны. Функция Empty(H) вырабатывает 1, если вход H пуст. Функция Search присваивает параметру по ссылке result значения (true,P), если нашли требуемый идентификатор, и P - указатель на

него, (false,NIL), если искомого идентификатора нет и в таблице нет свободного места, и (false,P), если искомого идентификатора нет, но в таблице есть свободный вход P.

Занесение идентификатора в таблицу можно осуществить следующей функцией:

```
int Insert(String id)
{boolean Yes;
  int Point;
 Search(id,&Yes, &Point);
 if (!Yes && (Point!=NULL))
   InsertId(Point,id);
 return(Point);
}
```

Здесь функция InsertId(Point,id) заносит идентификатор id по входу таблицы Point.

Второй способ организации таблицы идентификаторов - хранение идентификаторов в сплошном массиве символов. В этом случае идентификатору соответствует номер его первого символа в этом массиве, как это изображено на рис. 6.2. Идентификатор в массиве заканчивается каким-либо специальным символом (EOS). Второй возможный вариант - в качестве первого символа идентификатора в массив заносится его длина. Для организации поиска в таком массиве таблица идентификаторов отделяется от таблицы расстановки (рис. 6.3).

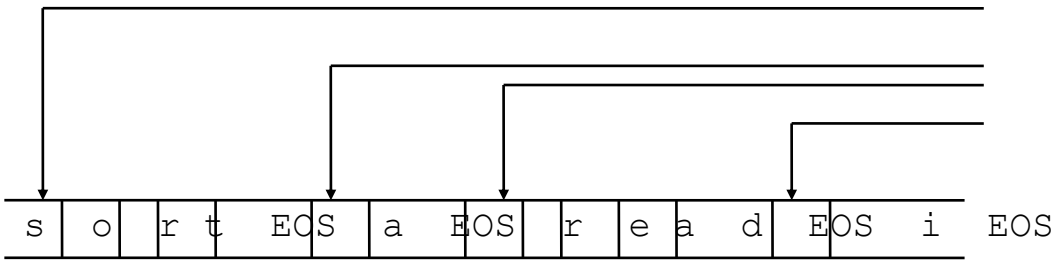


Рис. 6.2

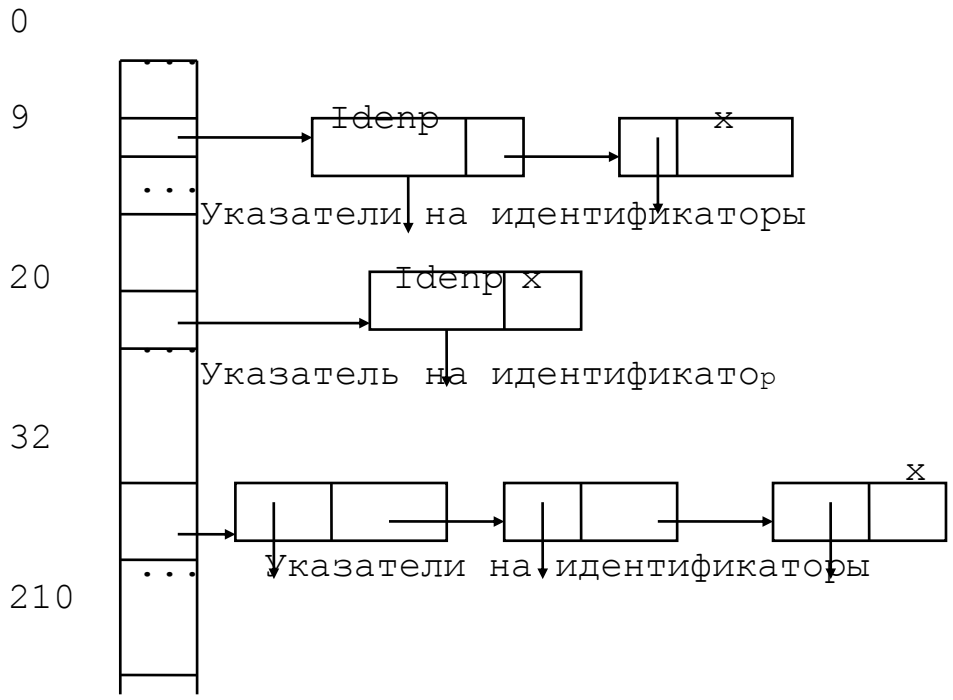


Рис. 6.3

6.3. Таблицы символов и таблицы расстановки

Рассмотрим организацию таблицы символов с помощью таблицы расстановки. Таблица расстановки - это массив указателей на списки указателей на идентификаторы. В каждый такой список входят указатели на идентификаторы, имеющие одно значение функции расстановки (рис. 6.3).

Вначале таблица расстановки пуста (все элементы имеют значение NIL). При поиске идентификатора id вычисляется функция расстановки H(id) и просматривается линейный список T[H]. Поиск в таблице может быть описан следующей процедурой:

```
struct    Element
   {String IdenP;
   struct Element * Next;
   };
Element * Search(String Id)
{Element * P;
  P=T[h(Id)];

  while (1)
   {if (P==NULL) return(NULL);
    else if (!IdComp(P->IdenP,Id)) return(P);
```

```
        else P=P->Next;
   }
}
```

IdenTab - таблица идентификаторов. Занесение объекта в таблицу может осуществляться следующей процедурой:

```
Element * Insert(Id:string)
{Element * P,H;
 P=Search(Id);
 if (P!=NULL) return(P);
 else    {H=H(Id); P=new Element();
       P->Next=T[H]; T[H]=P;
       P->Idenp=Include(Id);
      }
      return(P);
}
```

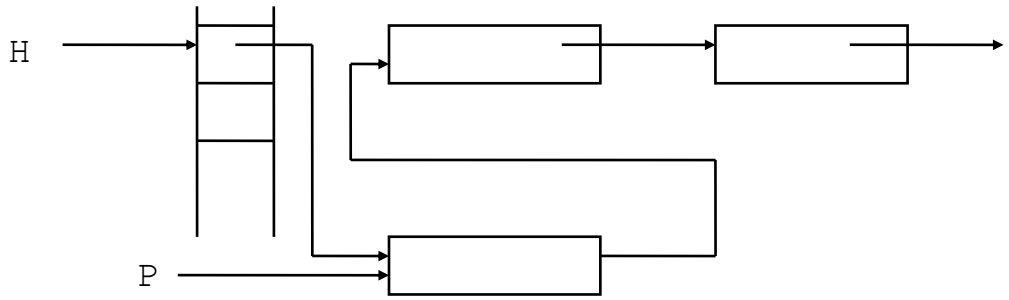


Рис. 6.4.

Процедура Include заносит идентификатор в таблицу идентификаторов. Алгоритм иллюстрируется рис. 6.4.

6.4. Функции расстановки.

Много внимания было уделено тому, какой должна быть функция расстановки. Основные требования к ней очевидны: она должна легко вычисляться и распределять равномерно. Один из возможных подходов 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 списка, т.е. целое между 0 и m-1, где m - размер таблицы расстановки, например, взятием остатка при делении H на m.

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 $1<=i<=k$, 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 [3], вычисляется, начиная с H=0. Для каждого символа с 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;
 }
```

Рис. 6.5

6.5. Таблицы на деревьях

Рассмотрим еще один способ организации таблиц с использованием

двоичных деревьев. Будем считать, что на множестве идентификаторов задан некоторый линейный порядок (например, лексикографический), т.е. задано некоторое отношение '<', транзитивное, антисимметричное и антирефлексивное. Каждой вершине двоичного дерева, представляющего таблицу символов, сопоставлен идентификатор. Вершина может иметь нуль, одного (правого или левого) или двух (правого и левого) потомков. Если вершина имеет левого потомка, то идентификатор, сопоставленный левому потомку, меньше идентификатора, сопоставленного самой вершине; если имеет правого потомка, то ее идентификатор больше. Элемент таблицы изображен на рис. 6.6.

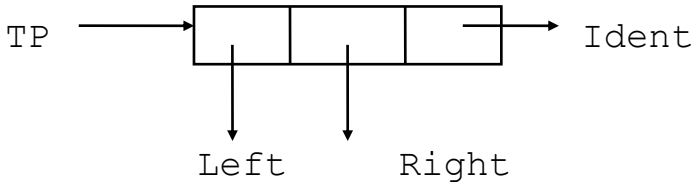


Рис. 6.6

```
struct TreElement
   {struct TreElement * Left, * Right;
    String IdenP;
    };
```

Поиск в такой таблице может быть описан следующей процедурой:

```
TreElement * SearchTree(String Id,TreElement * TP)
{int comp;
  if (TP==NULL) return NULL;
  comp= IdComp(Id,TP->IdenP);
  if (comp<0) return(SearchTree(Id,TP->Left));
  if (comp>0) return(SearchTree(Id,TP->Right));
  return TP;
}
```

Занесение в таблицу осуществляется функцией:

```
TreElement *  InsertTree(String Id,TreElement * TP);
TreElement * fill(TreElement * P, String Id)
{ if (P==NULL)
   {P=new TreElement();
    P->IdenP=Include(Id);
    P->Left=NULL;
    P->Right=NULL;
    return(P);
```

```
   }
    else return(InsertTree(Id,P));
}
TreElement *  InsertTree(String Id,TreElement * TP)
{int comp= IdComp(Id,TP->IdenP);
  if (comp<0) return(fill(TP->Left,Id));
  if (comp>0) return(fill(TP->Right,Id));
  return(TP);
}
```

Как показано в работе [7], среднее время поиска и занесения в таблицу размера n, организованную в виде двоичного дерева, при равной вероятности появления каждого объекта равно 2ln(2)log2(n)+C, что превышает на 40% среднее время поиска в упорядоченном массиве методом двоичного поиска. Это превышение обусловлено тем, что дерево неизбежно оказывается несбалансированным: имеются более длинные и более короткие ветви.

Чтобы ускорить 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 правого поддерева не отличается от высоты левого более чем на 1. Ясно, что для того, чтобы достичь сбалансированности, в процессе построения дерево иногда приходится слегка перестраивать [8]. Определим для каждой вершины дерева характеристику, равную разности высот правого и левого ее потомков. Для сбалансированного дерева характеристика вершины может быть равной -1, 0 и 1, для листьев она равна 0). Пусть мы определили место новой вершины в дереве. Ее характеристика равна 0. Рассмотрим отрезок пути от новой вершины к корню, такой, что характеристики всех вершин на нем равны 0 (до перестраивания). Пусть А - верхний (ближайший к корню) конец этого отрезка. А - либо корень, либо имеет характеристику 0. Если А - корень, то дерево перестраивать не надо, достаточно лишь изменить характеристики вершин на нем на 1 или -1, в зависимости от того, влево или вправо пристроена новая вершина.

Если верхний конец А отрезка с характеристикой 0 не корень, то возможны следующие варианты. Если А имеет характеристику 1 (-1) и новая вершина добавляется в правое (левое) поддерево, то характеристика вершины А становится равной 0 и дерево перестраивать не надо.

Пусть теперь характеристика А до перестраивания равна -1 и новая вершина добавлена к левому поддереву А (аналогично - для случая 1 и добавления к правому поддереву). Возможны следующие варианты.

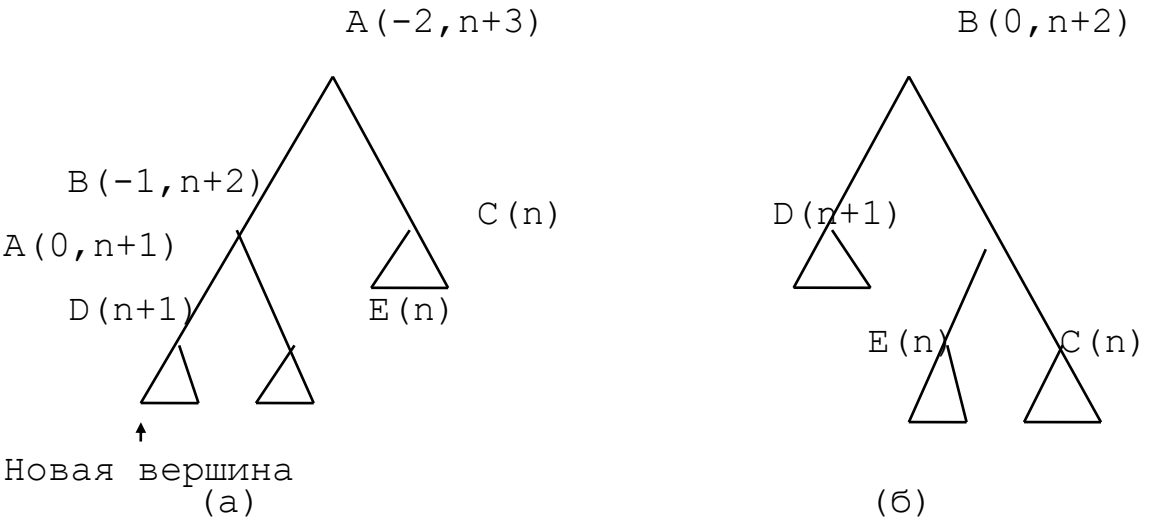

A(-2,n+3)
B(-1,n+2)
C(n)
A(0,n+1)
D(n+1)
E(n)
Новая вершина
(а)
B(0,n+2)
D(n+1)
E(n)
C(n)
(б)

Рис. 6.7

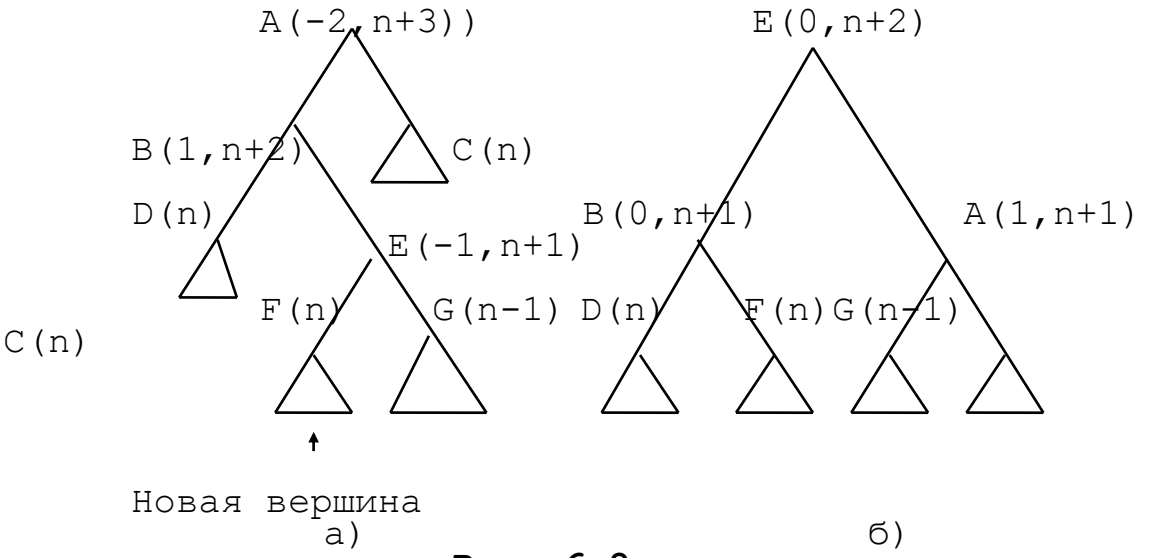

A(-2,n+3))
B(1,n+2)
C(n)
D(n)
E(-1,n+1)
F(n)
G(n-1)
C(n)
Новая вершина
а)
E(0,n+2)
B(0,n+1)
A(1,n+1)
D(n)
F(n)
G(n-1)
б)

Рис. 6.8

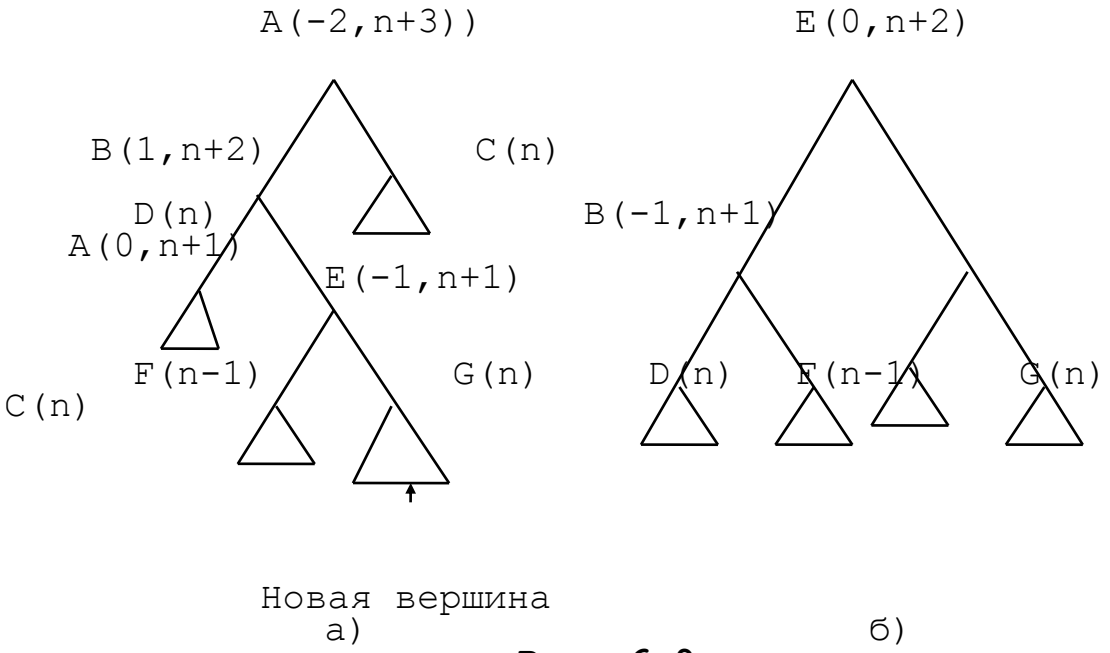


Рис. 6.9

Рассмотрим вершину B - левого потомка A. Если характеристика B после добавления новой вершины в D стала равна -1, то дерево имеет структуру, изображенную на рис. 6.7(а). Перестроив дерево так, как это изображено на рис. 6.7(б), мы добьемся сбалансированности (в скобках указаны характеристики вершин, где это существенно, и соотношения высот после добавления).

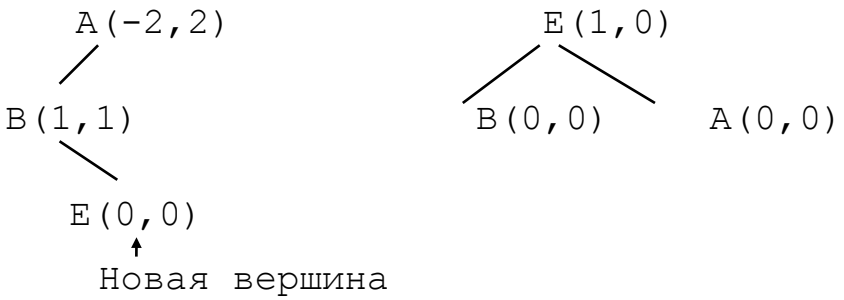


Рис. 6.10

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. 6.8, 6.9 и 6.10.

6.6. 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 объектов (и, возможно, одну общую таблицу идентификаторов). Работу с таблицами блоков можно организовать в магазинном режиме: при входе в блок создавать таблицу объектов, при выходе - уничтожать. При этом сами таблицы должны быть связаны в упорядоченный список, чтобы можно было просматривать их в порядке вложенности (рис. 6.11). Если таблицы организованы с помощью функций расстановки, это означает, что для каждой таблицы должна быть создана своя таблица расстановки.

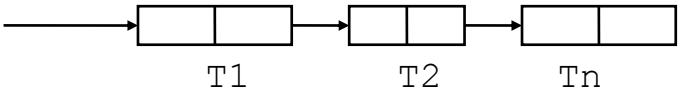


Рис. 6.11

6.7. Сравнение различных методов реализации таблиц

Рассмотрим преимущества и недостатки тех или иных методов реализации таблиц с точки зрения техники использования памяти. Если таблица размещается в массиве, то, с одной стороны, отпадает необходимость использования динамической памяти, а с другой - появляется ряд осложнений. Использование динамической памяти, как правило, довольно дорогая операция, поскольку механизмы поддержания работы с динамической памятью довольно 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 7. Промежуточные представления программы

В процессе трансляции программы часто используют промежуточное представление (ПП) программы, предназначенное прежде всего для удобства генерации кода и/или проведения различных оптимизаций программы. Сама форма ПП зависит от целей его использования. Наиболее часто используемыми формами ПП является ориентированный граф (или, в частности, абстрактное синтаксическое дерево), тройки, четверки, префиксная или постфиксная запись, атрибутированное абстрактное дерево.

7.1. Представление в виде ориентированного графа

Простейшей формой промежуточного представления является синтаксическое дерево программы. Более полную информацию о входной программе 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. 7.1.

На рис. 7.2. приведены два представления в памяти синтаксического дерева на рис. 7.1.а). Каждая вершина кодируется записью с полем для операции и полями для указателей на потомков. На рис. 7.2.б) вершины размещены в массиве записей и индексом (или входом) вершины служит указатель на нее.

7.2. 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 побитовая. В правую часть может входить только один знак операции.

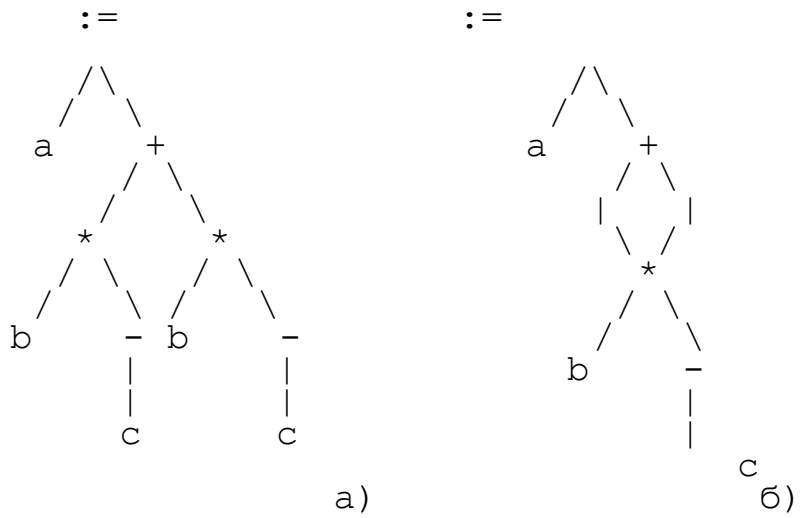
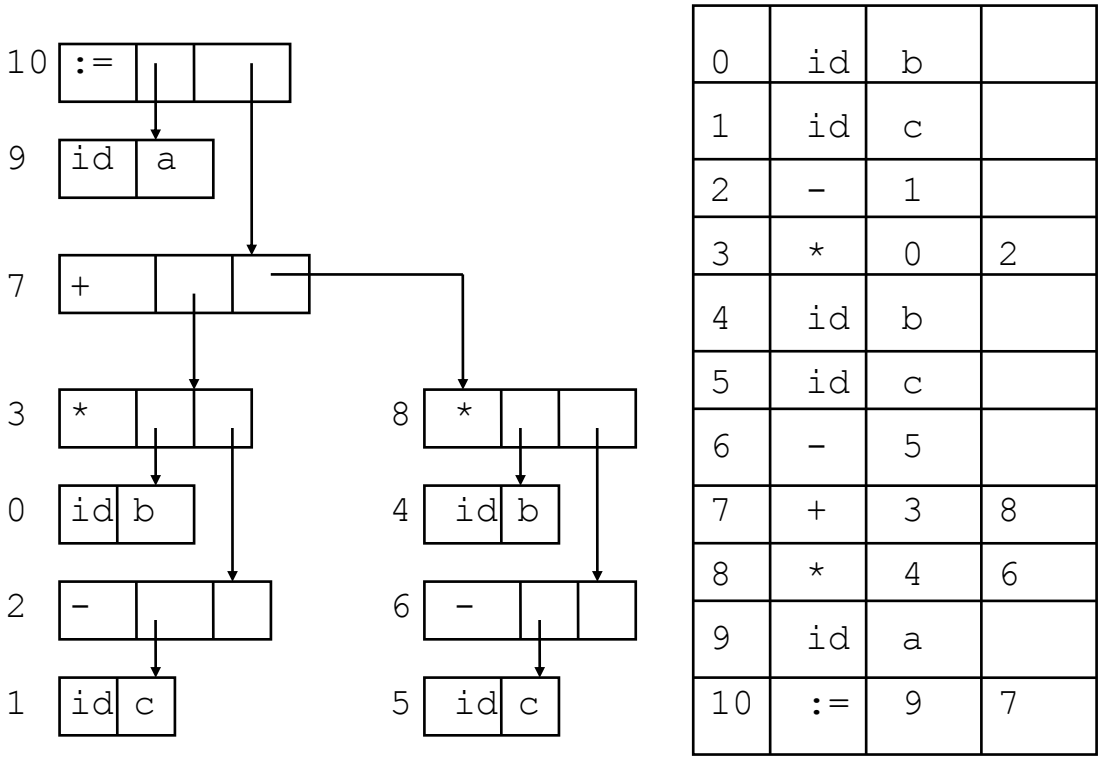


Рис. 7.1



0	id	b	
1	id	c	
2	-	1	
3	*	0	2
4	id	b	
5	id	c	
6	-	5	
7	+	3	8
8	*	4	6
9	id	a	
10	:=	9	7

Рис. 7.2

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 явные 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 + t1
a := t5

     а)                б)
```

Рис. 7.3

Представления синтаксического дерева и графа рис. 7.1 в виде трехадресного кода дано на рис. 7.3.а) и 7.3.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 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 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. 7.4 приведены четверки для a:=b*-c+b*-c. Они получены из трехадресного кода рис. 7.3.а.

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. 7.4.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. 7.4.б соответствуют четверкам рис. 7.4.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. 7.4

Для представления тройками трехместной операции типа x[i]:=y требуется два входа, как это показано на рис. 7.5.а, представление x:=y[i] двумя операциями показано на рис. 7.5.б.

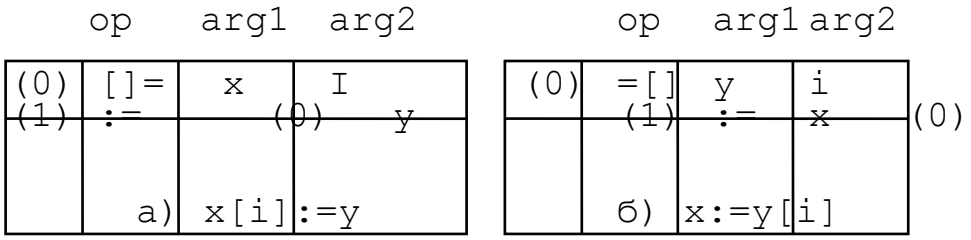


Рис. 7.5

Трехадресный код может быть представлен не списком троек, а списком указателей на них. Такая реализация обычно называется *косвенными тройками*. Например, тройки рис. 7.4.б могут быть реализованы так, как это изображено на рис. 7.6.

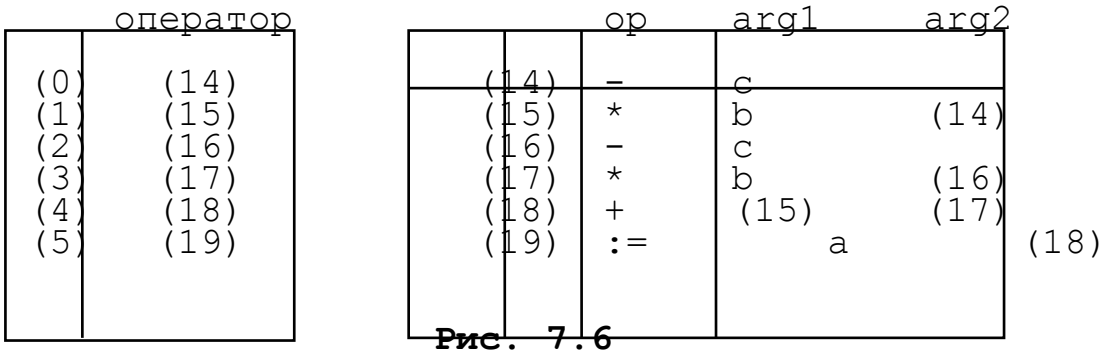


Рис. 7.6

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. 7.6 можно объединить строки (14) и (16), после чего можно объединить строки (15) и (17).

7.3. Линеаризованные представления

В качестве промежуточных представлений весьма распространены линеаризованные представления. Линеаризованное представление позволяет относительно легко хранить промежуточное представление на внешней памяти и обрабатывать его в порядке чтения. Самая распространенная форма линеаризованного представления - это запись дерева либо в порядке его обхода снизу-вверх (*постфиксная* запись, или *обратной польской*), либо в порядке обхода его сверху-вниз (*префиксная* запись, или *прямой польской*).

Таким образом, постфиксная запись (обратная польская) - это список вершин дерева, в котором каждая вершина следует непосредственно за своими потомками. Дерево на рис. 7.1 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 [4]. Лидер - это аббревиатура от 'ЛИнеаризованное ДЕРево'. Это машинно-независимая языково-ориентированная префиксная запись. В этом представлении сохраняются все объявления и каждому из них присваивается свой уникальный номер, который используется для ссылки на объявление. Рассмотрим пример (рис. 7.7).

```
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.
```

Рис. 7.7

Соответствующий образ в Лидере изображен на рис. 7.8.

```
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
```

Рис. 7.8

Рассмотрим его более детально:

```
program 'M'   Имя модуля используется для редактора
              связей
var int       Это образ переменных var X,Y,Z:integer;
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       Локальная 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 тип целый
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 и Y
int var 0 2 end
end           Конец вызова
end           Конец тела модуля
```

7.4. 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 костанты или переменной, их предварительно необходимо загрузить на верхушку стека. Код операции - всегда один байт. Если операция имеет операнды, они располагаются в следующих байтах.

Элементарные типы данных, с которыми работает машина, - short, integer, long, float, double, все знаковые.

7.4.1. Организация памяти

Машина имеет следующие регистры:

pc - счетчик команд;

optop - указатель вершины стека операций;

frame - указатель на стэк-фрейм исполняемого метода;

vars - указатель на 0-ю переменную исполняемого метода.

Все регистры 32 разрядные. Других регистров нет.

Стэк-фрейм имеет три компоненты:

локальные переменные,

среда исполнения,

стэк операндов.

Локальные переменные отсчитываются от регистра vars.

Среда исполнения служит для поддержания самого стэка. 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 исползуется методами.

7.4.2. Набор команд виртуальной машины

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.
Передачи управления.
Возврат из функции.
Табличный переход.
Обработка полей объектов.
Вызов метода.
Обработка исключительных ситуаций.
Прочие операции над объектами.
Мониторы.
Отладка.

Рассмотрим некоторые группы команд подробнее.

Помещение локальных переменных на стек.

iload - загрузить целое из локальной переменной.

Операндом операции 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.

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

invokevirtual

При трансляции объектно ориентированных языков программирования из-за возможности перекрытия виртуальных методов вообще говоря статически нельзя 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 implrments 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 область 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 etc.) и число аргументов, ожидаемых на стеке операндов.

Если метод помечен как synchronized, запускается монитор, связанный с handle. Базис массива локальных переменных для нового стэк фрейм устанавливается так, что он указывает на handle на стеке, так что handle и первые локальные переменные нового фрейма параметрами вызова. Определяется общее число локальных переменных, используемых методом, и после того, как отводится 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. Наконец исполнение продолжается с первой инструкции вызванного метода.

Обработка исключительных ситуаций

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 обрабатывет, и с ним связан набор инструкций, которые его реализуют. При возникновении исключительной ситуации просматривается список операторов catch, чтобы установить соответствие. Исключительная ситуация соответствует оператору catch, если инструкция, вызавшая исключительную ситуацию, лежит в соответствующем диапазоне и исключительная ситуация принадлежит подтипу типа ситуации, которые обарабатывает оператор catch. Если соответствующий оператор catch найден, управление передается обработчику. Если нет, текущий стэк-фрейм удаляется, и и сключительная ситуация возбуждается вновь.

Порядок операторов catch в списке важен. Интерпретатор передают управление первому подходящему оператору catch.

7.5. Организация информации в генераторе кода

Чисто синтаксическое дерево 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. 1) всю эту информацию можно хранить в таблицах генератора кода; 2) информация хранится в вершинах дерева с соответствующими указателями.

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.). Таким образом структура таблиц может быть такой, как это изображено на рис. 7.9.

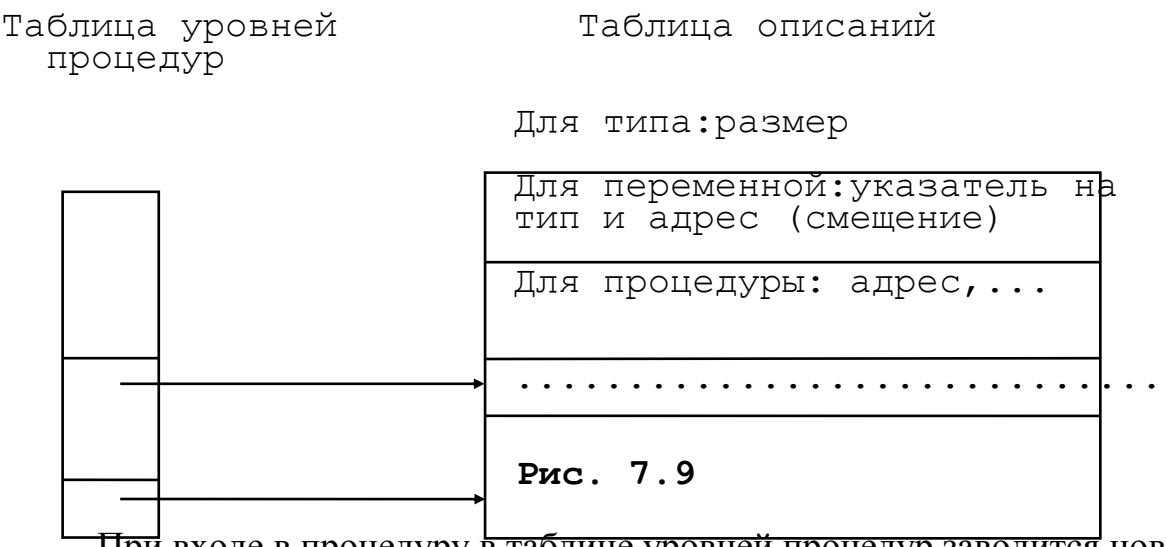


Рис. 7.9

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. Например, та же информация может быть представлена, как на рис. 7.10.

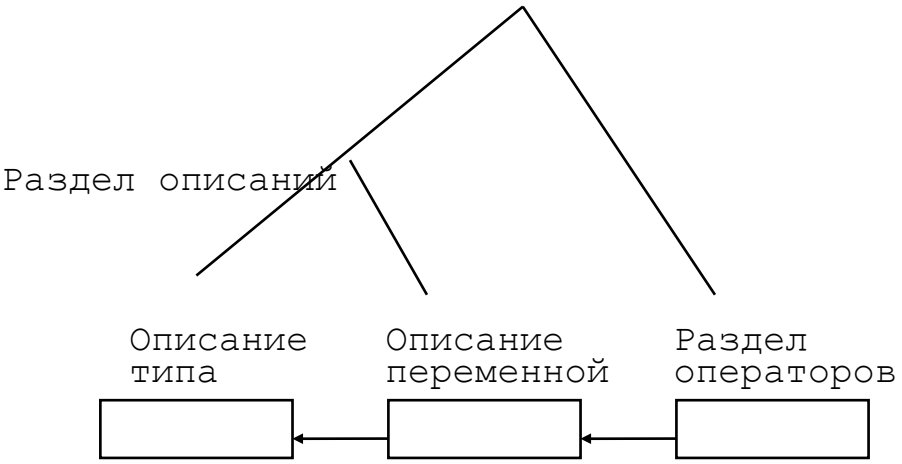


Рис. 7.10

7.6. 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 лучше годятся для различного рода оптимизаций (например, косвенные тройки - для перемещения кода), некоторые - хуже (например, префиксная запись для этого плохо подходит).

Глава 8. Генерация кода

Задача генератора кода - построение эквивалентной машинной программы по программе на входном языке. Обычно в качестве входного для генератора кода служит некоторое промежуточное представление программы. В свою очередь, генерация кода состоит из ряда специфических, относительно независимых подзадач: распределение памяти (в частности, распределение регистров), выбор команд, генерация 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) необходимо ведет к генерации той или иной последовательности команд. Однако удобно и практично эти задачи все же разделять и при этом особо обращать 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 и 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 и алгоритмы генерации кода будем излагать в виде атрибутных схем со входным языком Лидер.

8.1. Модель машины

При изложении алгоритмов генерации кода мы будем следовать некоторой модели машины, в основу которой положена система команд микропроцессора Motorola 68020.

В системе команд используются следующие способы адресации:

D - значение находится в регистре данных;

A - значение находится в адресном регистре;

POST - пост-инкрементная адресация (A)+: исполнительный адрес есть значение адресного регистра и после исполнения команды значение этого регистра увеличивается на длину операнда;

PRE - пре-инкрементная адресация -(A): перед исполнением операции содержимое адресного регистра уменьшается на длину операнда, исполнительный адрес равен новому содержимому адресного регистра;

DIRECT - прямая адресация через регистры: исполнительный адрес равен значению выражения ADDREG+INDEXREG*SCALE +ADDRDISP - содержимое адресного регистра + содержимое индексного регистра, умноженное на SCALE+адресное смещение;

INDPPRE - пре-косвенная через память: (ADDREG+INDEXREG* SCALE+ADDRDISP)+INDEXDISP - выражение в скобках дает адрес ячейки, содержимое которой + индексное смещение дает исполнительный адрес;

INDPOST - пост-косвенная через память: (ADDREG+ADDRDISP) +INDEXREG*SCALE+INDEXDISP - выражение в скобках дает адрес ячейки, содержимое которой + содержимое индексного регистра, умноженное на SCALE + индексное смещение, дает исполнительный адрес;

DIRPC - прямая через PC (счетчик команд): исполнительный адрес определяется выражением PC+INDEXREG*SCALE+ADDRDISP;

INDPREPC - пре-косвенная через PC: (PC+INDEXREG* SCALE+ ADDRDISP)+INDEXDISP - то же, что INDPRE, но в качестве адресного регистра используется PC;

INDPOSTPC - пост-косвенная через PC: (PC+ADDRDISP)+INDEXREG *SCALE+INDEXDISP то же, что и INDPOST, но в качестве адресного регистра используется PC;

ABS - абсолютная;

IMM - непосредственный операнд.

В дальнейшем изложении будут использованы следующие команды:

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 по адресу ИА;

MOVEM ИА,список регистров - восстановить указанные регистры из памяти по адресу ИА;

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;

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;

CMP ИА,D - содержимое регистра данных D вычитается из содержимого по исполнительному адресу ИА, при этом формируется признак результата, но содержимое регистра D не меняется;

TST ИА - выработать код условия по значению, находящемуся по исполнительному адресу ИА;

BNE ИА - условный переход по признаку NE (не равно) по исполнительному адресу ИА;

BEQ ИА - условный переход по признаку EQ (равно) по исполнительному адресу ИА;

BLE ИА - условный переход по признаку LE (меньше или равно) по исполнительному адресу ИА;

BGT ИА - условный переход по признаку GT (больше) по исполнительному адресу ИА;

BLE ИА - условный переход по признаку KE (меньше или равно) по исполнительному адресу ИА;

BLT ИА - условный переход по признаку LT (меньше) по исполнительному адресу ИА;

BR ИА - безусловный переход по адресу ИА;

JSR ИА - переход с возвратом на подпрограмму 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.

8.2. Динамическая организация памяти

Рассмотрим схему реализации 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 определения процедур. Еще одной задачей, которую необходимо решать при трансляции языков с блочной структурой - обеспечение реализации механизмов статической вложенности. Рассмотрим рекурсивную программу рис. 8.1.

```
PROCEDURE P1;
   VAR V1;
   PROCEDURE P2;
      VAR V2;
      BEGIN P2;
            V1:=...
            V2:=...
      END P2;
   BEGIN P2;
   END P1;
```

Рис. 8.1

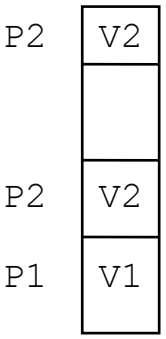


Рис. 8.2

В процессе выполнения этой программы в магазине может сложиться ситуация, изображенная на рис. 8.2. Находясь в процедуре P2, мы должны иметь доступ к последнему экземпляру значений переменных процедуры P2 и к экземляру 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). Использование той или иной схемы определяется, помимо прочих условий, прежде всего числом адресных регистров.

8.2.1. Организация магазина со статической цепочкой

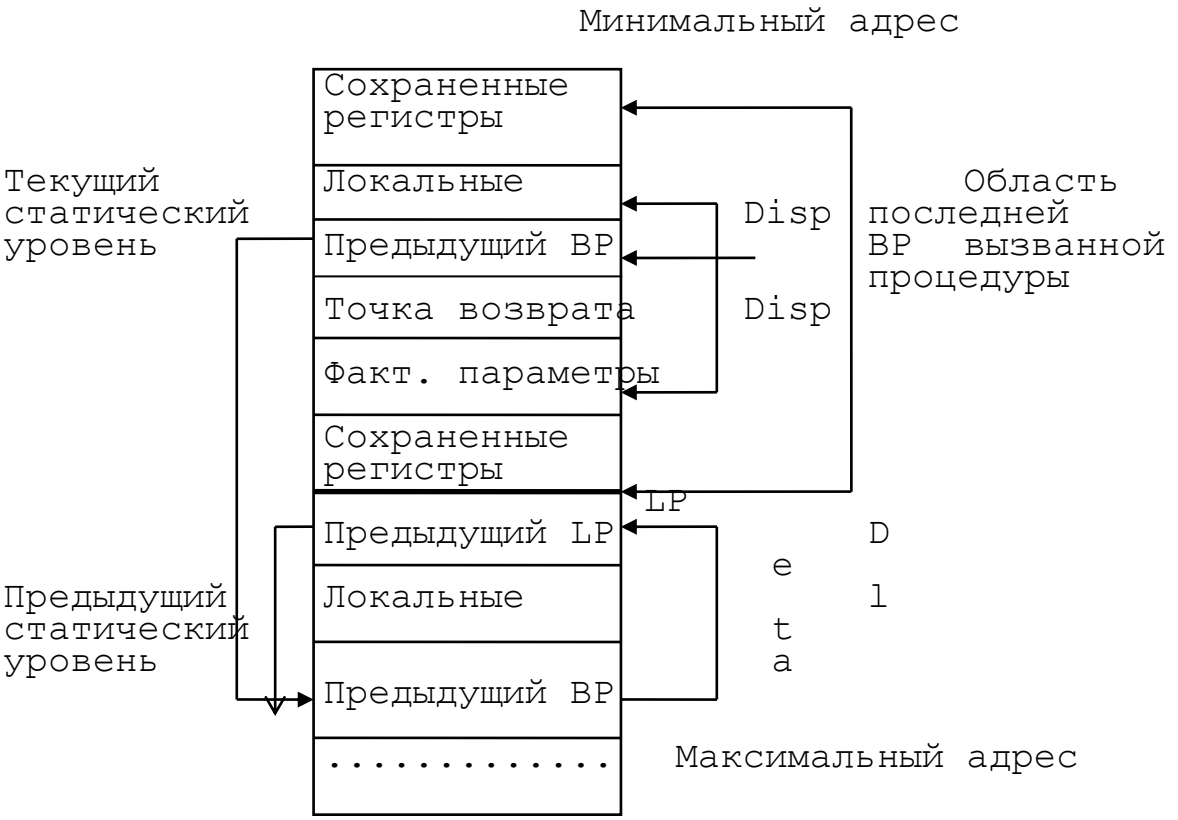


Рис. 8.3

Итак, в случае статической цепочки магазин организован, как это изображено на рис. 8.3.

Таким образом, на запись текущей процедуры в магазине указывает регистр BP (Base pointer), с которого начинается динамическая цепочка. 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 уровня производится несколько по-разному. При вызове подпрограммы того же статического уровня, что и вызывающая подпрограмма, 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. 8.4. Занесение BP, отведение локальных, сохранение регистров делает вызываемая подпрограмма.

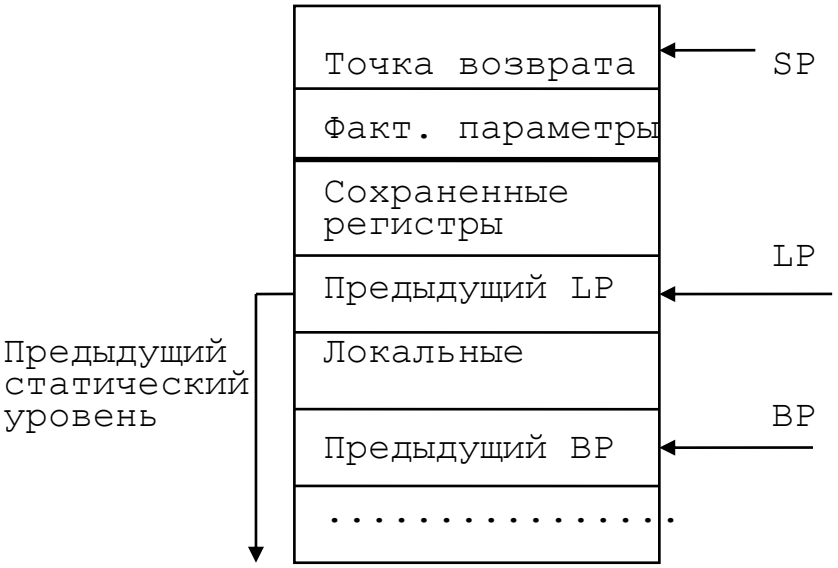


Рис. 8.4

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 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. 8.5. 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.

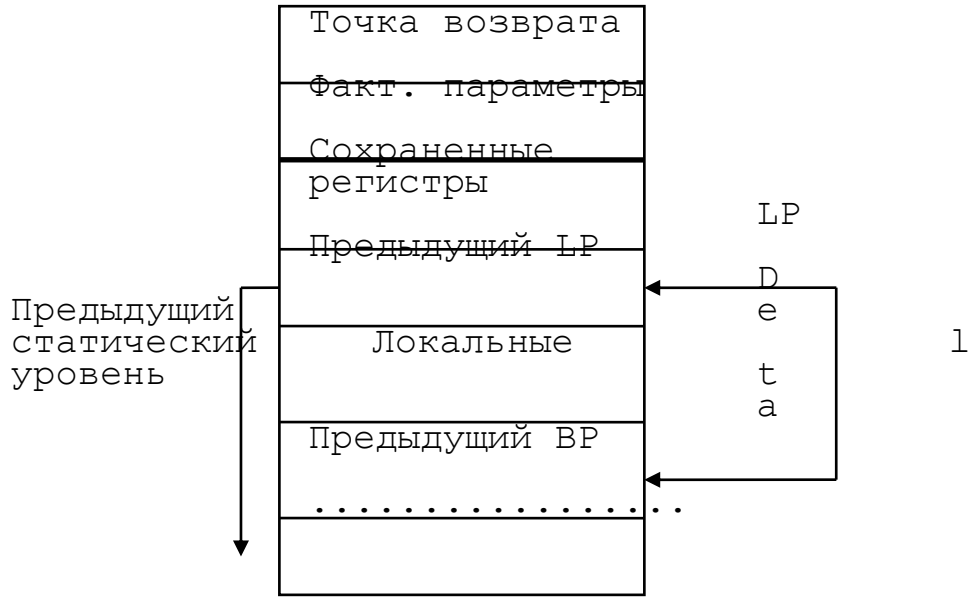


Рис. 8.5

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
```

восстанавиливающая 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. 8.3.

Выход из подпрограммы осуществляется следующей последовательностью команд:

```
MOVEM (SP)+
UNLK BP
RTD размер фактических
```

Команда MOVEM восстанавливает регистры из магазина. После ее выполнения магазин выглядит, как это изображено на рис. 8.6.

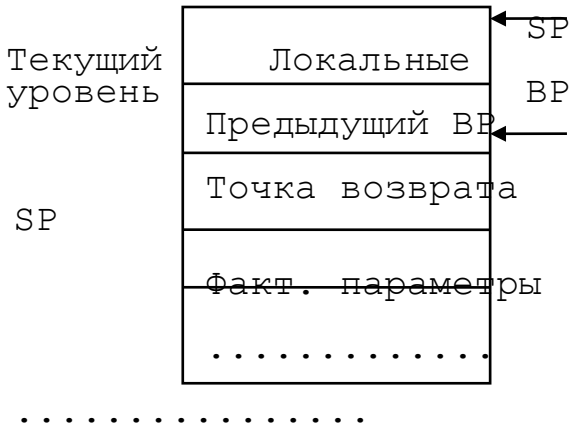


Рис. 8.6

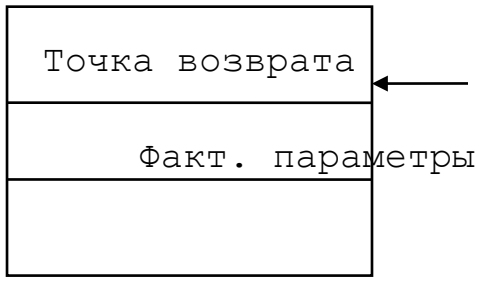


Рис. 8.7

После выполнения команды UNLK BP, которая эквивалентна такой последовательности команд:

```
MOVE BP,SP
MOVE (SP),BP
ADD #4,SP /*4 - размер слова*/
```

магазин имеет содержимое, изображенное на рис. 8.7.

Наконец, после выполнения команды RTD размер_фактических, которая эквивалентна последовательности

```
ADD размер_фактических+4, SP
JMP -размер_фактических-4(SP)
```

магазин восстанавливается до состояния, которое было до вызова.

В зависимости от наличия локальных переменных, фактических параметров и необходимости упрятывания регистров каждая из этих команд может отсутствовать.

8.2.2. Организация магазина с дисплеем

Рассмотрим теперь организацию магазина с дисплеем. Дисплей - это

массив (DISPLAY), i-й элемент которого представляет собой указатель на область активации процедуры i-го статического уровня. Доступ к переменным самой внутренней процедуры осуществляется через регистр BP. При вызове процедуры меньшего статического уровня изменений в дисплее не производится. При вызове локальной процедуры в дисплее отводится элемент для текущей (вызывающей) процедуры. При вызове процедуры того же уровня (i) i-й элемент замещается на указатель области активации текущей процедуры и при выходе восстанавливается. Тем самым DISPLAY[i] всегда указывает на область активации последней вызванной процедуры i-го статического уровня.

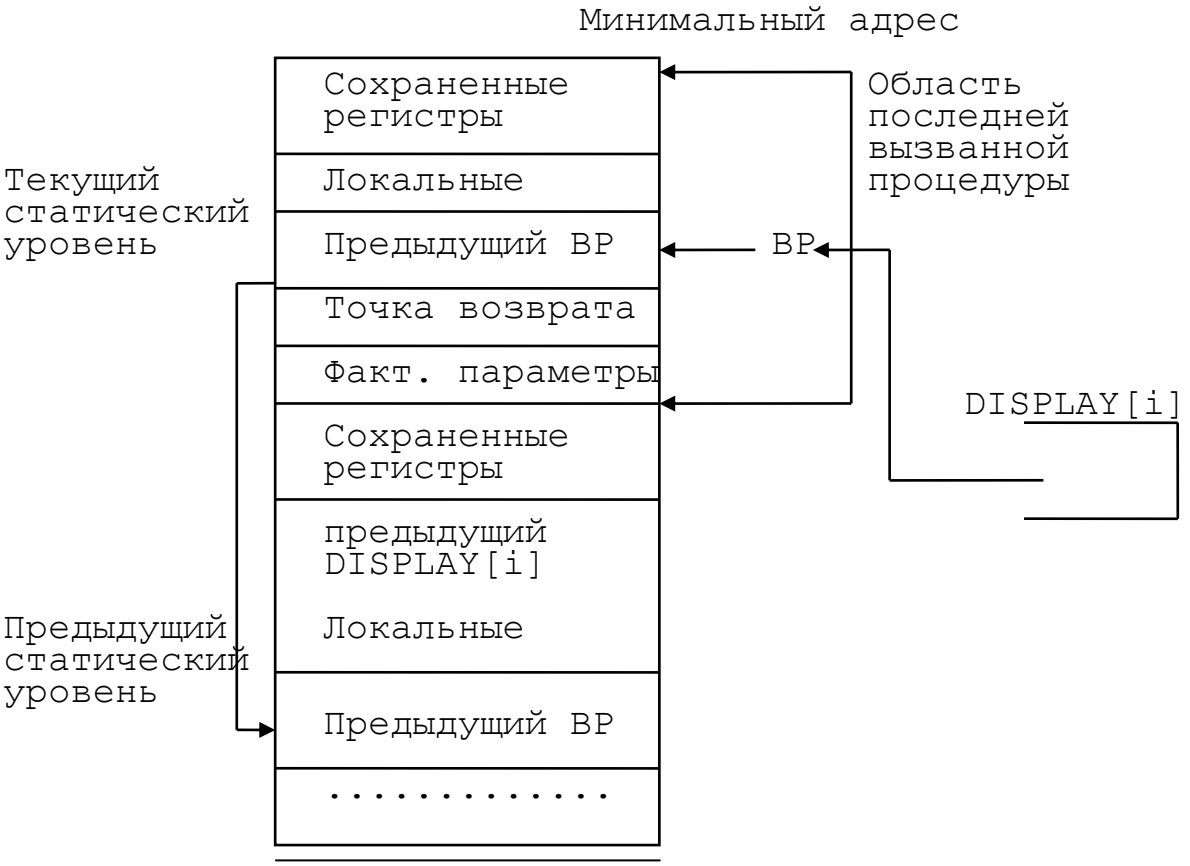


Рис. 8.8

Запоминание и восстановление DISPLAY[i] можно не выполнять, если процедура не имеет локальных переменных и параметров или если из процедуры нет вызовов описанных в ней процедур. Структура магазина при использовании дисплея изображена на рис. 8.8. 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<=i элементы дисплея j,…,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 содержимой 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.

8.3. 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 - адрес;
- для поля записи - смещение внутри записи;
- для процедуры - размер локальных;
- для диапазона индексов массива - значение левой границы.

Функция IncTab вырабатывает указатель (вход) на новый элемент таблицы, проверяя при этом наличие свободной памяти.

Таблица LevelTab - это таблица уровней процедур, содержащая указатели на последовательно вложенные процедуры (см. рис. 4.9).

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>.
```

Это можно следующим образом изобразить на дереве объявлений (рис. 8.10).

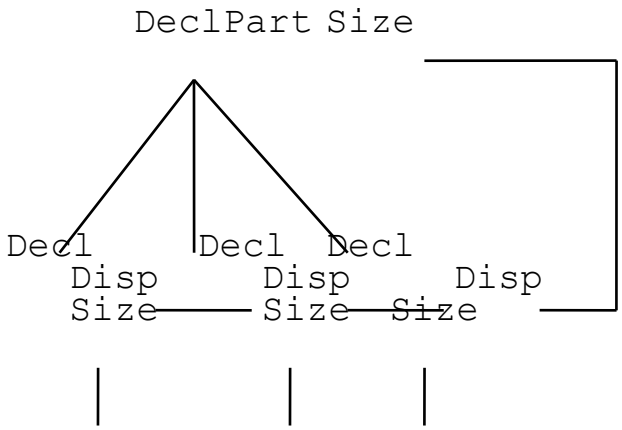


Рис. 8.10

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.
```

8.4. 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,NO};
enum AddrMode
    {D,A,Post,Pre,Direct,IndPre,IndPost,
              DirPC,IndPrePC,IndPostPC,Abs,Imm};
 struct AddrType{
   Register Addreg,IndexREG;
   int IndexDisp,AddrDisp;
   short Scale;
};
```

Значение регистра NO 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 процедуры 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;
AddrType AddrTmp1,AddrTmp2;
```

```
3: if (Val<2>==0) // Глобальная переменная
      {Address<4>.AddrMode=Abs;
            Address<4>.AddrDisp=0;
      }
   else // Локальная переменная
        {Address<4>.AddrMode=Direct;
        if (Val<2>==Level<Block>)
           // Переменная текущего уровня
           Address<4>.Addreg=A6;
        else if (Val<2>==Level<Block>-1)
             // Переменная предыдущего уровня
              Address<4>.Addreg:=A5;
        else
{Address<4>.Address<4>.Addreg=GetFree(RegSet<Block>);
       AddrTmp1.AddrMode=Direct;
      AddrTmp1.Addreg=A5;
      AddrTmp1.IndexReg=No;
      AddrTmp1.AddrDisp=0;
      Emit2(MOVEA,AddrTmp1,Address<4>.Addreg);
      AddrTmp1.Addreg=Address<4>.Addreg;
      AddrTmp2.AddrMode=A;
      AddrTmp2.Addreg=Address<4>.Addreg;
             for (Temp=Level<Block>-Val<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 Address и служат операндами команды (здесь для упрощения изложения в качестве параметров этой процедуры используются контрукции типа '(A)'; имеется в виду косвенная адресация через адресный регистр). Смещение переменной текущего уровня отсчитывается от базы (A6), а других уровней - от указателя статической цепочки, поэтому оно определяется как алгебраическая сумма LOCALSize и величины смещения переменной (отрицательного!).

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=Direct;
        if (Val<2>=Level<Block>)
           // Переменная текущего уровня }
           {Address<4>.Addreg=A6;
           Address<4>.AddrDisp=Table[Val<3>];
       }
         else {Address<4>.AddrMode=IndPost;
       Address<4>.Addreg=NO;
       Address<4>.IndexREG=NO;
       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==Direct)
         Address<3>.AddrDisp=Address<0>.AddrDisp
                              +Table[Val<2>];

   else Address<3>.IndexDisp=Address<0>.IndexDisp
                               +Table[Val<2>];
   }.
```

Смещение в случае абсолютной и прямой адресации определяется полем AddrDisp, а в остальных случаях - полем IndexDisp. Таким образом, видно, что при обработке полей записей идет только накопление смещения поля

без генерации каких-либо команд.

Атрибутные зависимости для этого правила могут быть изображены следующим образом (рис. 8.11):

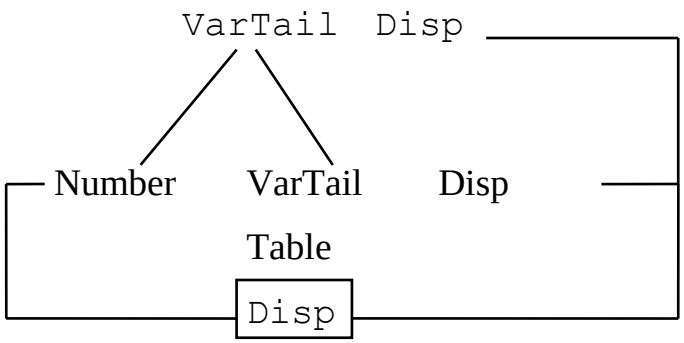


Рис. 8.11

Рассмотрим теперь выборку элемента массива. Лидер-синтаксис обращения к элементу массива следующий:

VarTail ::= 'ARR' Number Typexpr VarTail

Здесь Number - Лидер номер описания диапазона индексов; Typexpr - индексное выражение. По адресу левой части правила, представляющей переменную-массив, от которого берется индекс, и по индексному выражению, представленному нетерминалом Typexpr, мы должны сформировать адрес элемента массива. В приведенной ниже таблице решений функция GetAddr выбирает очередной свободный адресный регистр.

Под "рабочим" регистром в таблице решений имеется в виду регистр, значение которого может быть испорчено (примером "не рабочего" регистра может служить регистр A6 или регистр A5, если он используется как указатель процедуры предыдущего статического уровня).

MaxReg - максимальный номер адресного регистра, которым можно пользоваться как рабочим. Все адресные регистры разделены на два множества: имеющие номера, большие MaxReg, заняты предварительным распределением, остальные - рабочие. Функция GetAddr выдает очередной свободный рабочий адресный регистр.

Тип адресации	VarTail левой части: IndPre || Direct	(IndPost || Abs) || IndexReg==NO	((Direct || IndPre) && (IndReg!=NO))
ElSize<3>==1,2,4,8 AddrMode<3>==D AddrMode<4>=IndPost (Addreg<0>==рабочий) ?Addreg<0> :GetAddr	AddrDisp<4>:= AddrDisp<0> -Left<2>*ElSize<3> Addreg<4>=Addreg<0> IndexReg<4>= Addreg<3> AddrMode<4>=IndPost Scale<4>=ElSize<3>	AddrDisp<4>= - Addreg<4>= ?Addreg<0> :GetAddr IndexReg<4>= Addreg<3> Scale<4>=ElSize<3> LEA Address<0>, Address<4>	
ElSize<3> !=1,2,4,8 AddrMode<3>==D (Addreg<0>==рабочий) ?Addreg<0> :GetAddr	AddrDisp<4>= AddrDIP<0>- Left<2>*ElSize<3> Addreg<4>=Addreg<0> IndexReg<4>= Addreg<3> Scale<4>=1 AddrMode<4>=IndPost MUL ElSize<3>, Addreg<3>	AddrDisp<4>= -Left<2>*ElSize<3> AddrMode<4>=IndPost Addreg<4>= IndexReg<4>= Addreg<3> Scale=1 LEA Address<0>, Address<4> MUL ElSize<4>, IndexReg<4>	

ElSize<3>== 1,2,4,8 AddrMode<3>!=D	AddrDisp<4>= AddrDisp<0>- Left<2>*ElSize<3> AddrMode<4>=IndPost Addreg<4>=Addreg<0> IndexReg<4>=GetFree (Addreg<0>==рабочий) AddrMode<4>=IndPost ?Addreg<0> Scale<4>=ElSize<3> :GetAddr MOVE Address<3>, IndexReg<4>	AddrDisp<4>= -Left<2>*ElSize<3> AddrMode<4>=IndPost IndexReg<4>=GetFree() Scale<4>=ElSize<3> LEA Address<0>, Address<4> MOVE Address<3>, IndexReg<4>

ElSize<3> <>1,2,4,8 AddrMode<3><>D	AddrDisp<4>= AddrDisp<0>- Left<2>*ElSize<3> Addreg<4>=Addreg**<0>** IndexReg<4>=GetFree AddrMode<4>=IndPre Scale<4>=ElSize**<3>** MOVE Address<3>, IndexReg<4> MUL ElSize<3>, IndexReg<4>	AddrDisp<4>:= -Left<2>*ElSize AddrMode<4>=IndPre Addreg<4>= (Addreg<0>==рабочий) ?Addreg<0> :GetAddr IndexReg<4>=GetFree() Scale<4>=1 LEA Address<0>, Address<4> MOVE Address<3>, IndexReg<4> MUL ElSize<3>, IndexReg<4>

Приведенная таблица реализуется следующим правилом:

```
RULE
VarTail ::= 'ARR' Number Typexpr VarTail
SEMANTICS
 int Size;
 bool Flag;
AddrType AddrTmp1,AddrTmp2;

Flag= ((Address<0>.AddrMode==IndPre)
    || (Address<0>.AddrMode==Direct))
    && (Address<0>.IndexReg==NO);

Size=Table[Val<2>];
if (Address<3>.AddrMode==D)
```

```
      IndexReg<4>=Addreg<3>;
 else IndexReg<4>=GetFree;
AddrMode<4>=IndPre;
if (Flag)
  {Address<4>.AddrDisp=Address<0>.AddrDisp-Table[Val<2>]*Size;
  Addreg<4>=Addreg<0>;
  }
else {Address<4>.AddrDisp=-Table[Val<2>]*Size;
    if (Address<0>.Addreg<=MaxReg)
      Addreg<4>=Addreg<0>;
    else Addreg<4>=GetAddr;
      }
if (Size & 14) // 2,4,8
      Address<4>.Scale=Size);
else Address<4>.Scale=1;
AddrTmp1.AddrMode=D;
AddrTmp1.Addreg=IndexReg<4>;
if (Flag) Emit2(LEA,Address<0>,Address<4>);
if (Address<3>.AddrMode!=D)
 Emit2(MOVE,Address<3>,AddrTmp1);
AddrTmp2.AddrMode=IMM;
AddrTmp2.AddrDisp=ElSize<3>;
if (!(Size & 15)) // 1,2,4,8
 Emit2(MUL,AddrTmp2,AddrTmp1).
```

8.5. 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 регистрах, 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.8.12.

		Правый операнд A2 R	Правый операнд A2 V
Левый операнд A1	R	ADD A1,A2	ADD A2,A1
	V	ADD A1,A2	MOVE A1,R ADD A2,R

Рис. 8.12

Здесь имеется в виду, что R - операнд на регистре, V - операнд - 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>);
         }.
```

8.6. Распределение регистров при вычислении арифметических выражений

Одной из важнейших задач при генерации кода является распределение регистров. Рассмотрим хорошо известную технику распределения регистров при трансляции арифметических выражений, называемую алгоритмом Сети-Ульмана.

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.

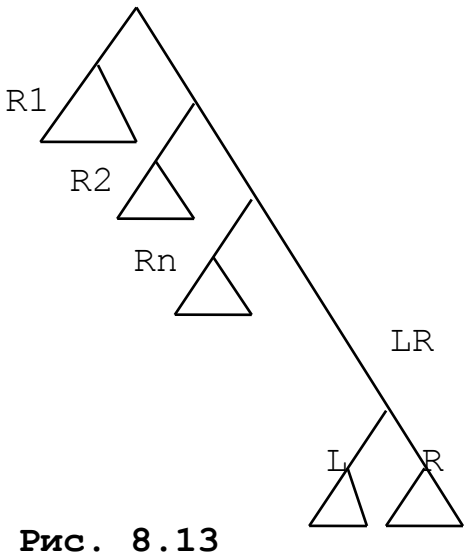


Рис. 8.13

Предположим сначала, что распределение регистров осуществляется по простейшей схеме слева-направо, как изображено на рис. 8.13. 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 распределении 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. 8.14).

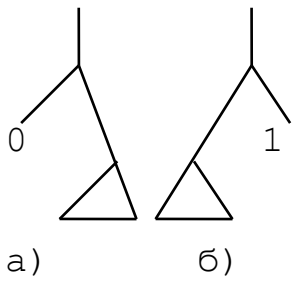


Рис. 8.14

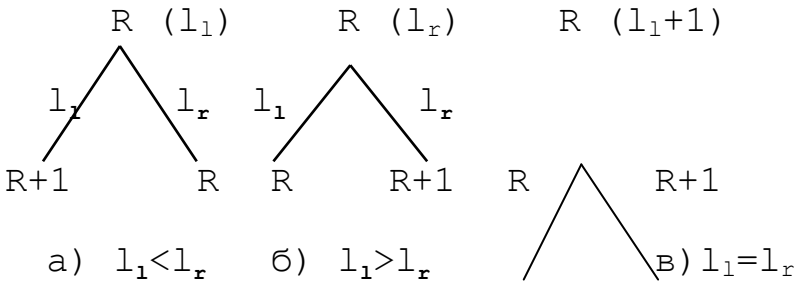


Рис. 8.15

2) если вершина имеет прямых потомков с метками l_1 и l_2, то в качестве метки этой вершины выбираем большее из чисел l_1 или l_2 либо число l_1+1, если $l_1=l_2$ (рис. 8.15).

Эта разметка позволяет определить, какое из поддеревьев требует большего количества регистров для своего вычисления.

Затем осуществляется распределение регистров для результатов операций.

Правила распределения регистров.

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, сначала вычисляется левое поддерево и его результат помещается в регистр R (рис. 8.15), а правому потомку – R+1. После этого формируется код по следующим правилам.

Правила генерации кода:

1) если вершина - правый лист с меткой 1, то ей соответствует код LOAD X,R, где R - регистр, назначенный этой вершине, а X - адрес переменной, связанной с вершиной (рис. 8.16.б);

2) если вершина внутренняя и ее левый потомок - лист с меткой 0, то ей соответствует код (рис. 8.16а)

```
Код правого поддерева
Op X,R
```

где снова R - регистр, назначенный этой вершине, X - адрес переменной, связанной с вершиной, а Op - операция, примененная в вершине (рис. 8.16.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. 8.17 б)).

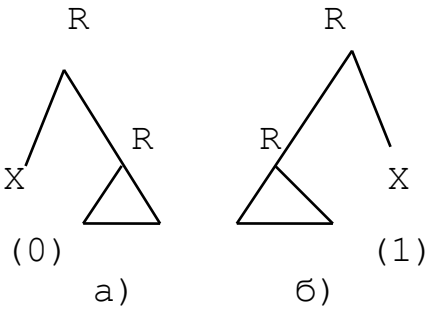


Рис. 8.16

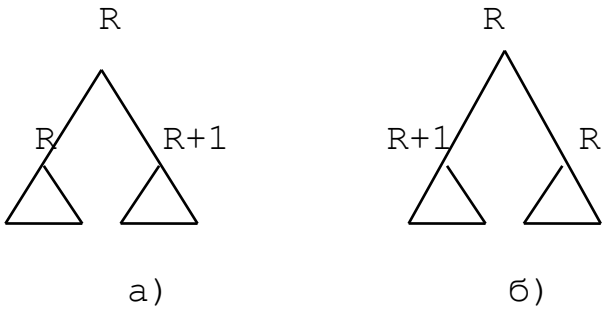


Рис. 8.17

Если 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. 8.17 а)).

Рассмотрим атрибутную грамматику, реализующую эти правила генерации кода. В этой атрибутной грамматике 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; 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>
             + Reg<3> + "," + 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)
RULE
IntExpr ::= Ident
SEMANTICS
Label<0>=(Left<0>) ? 0 : 1;
Code<0>=(!Left<0>)
```

```
   ? "LOAD" + Reg<0> + "," + Val<1>
   : Val<1>.

RULE
Op ::= '+'
SEMANTICS
Code<0>="ADD".

RULE
Op ::= '-'
Code<0>="SUB".

RULE
Op ::= '*'
SEMANTICS
Code<0>="MUL".

RULE
Op ::= '/'
SEMANTICS
Code<0>="DIV".
```

Атрибутированное дерево для выражения A*B+C*(D+E) приведено на рис. 8.18. При этом будет сгенерирован следующий код:

```
LOAD E,R1
ADD D,R1
MUL C,R1
LOAD B,R2
MUL A,R2
ADD R2,R1
```

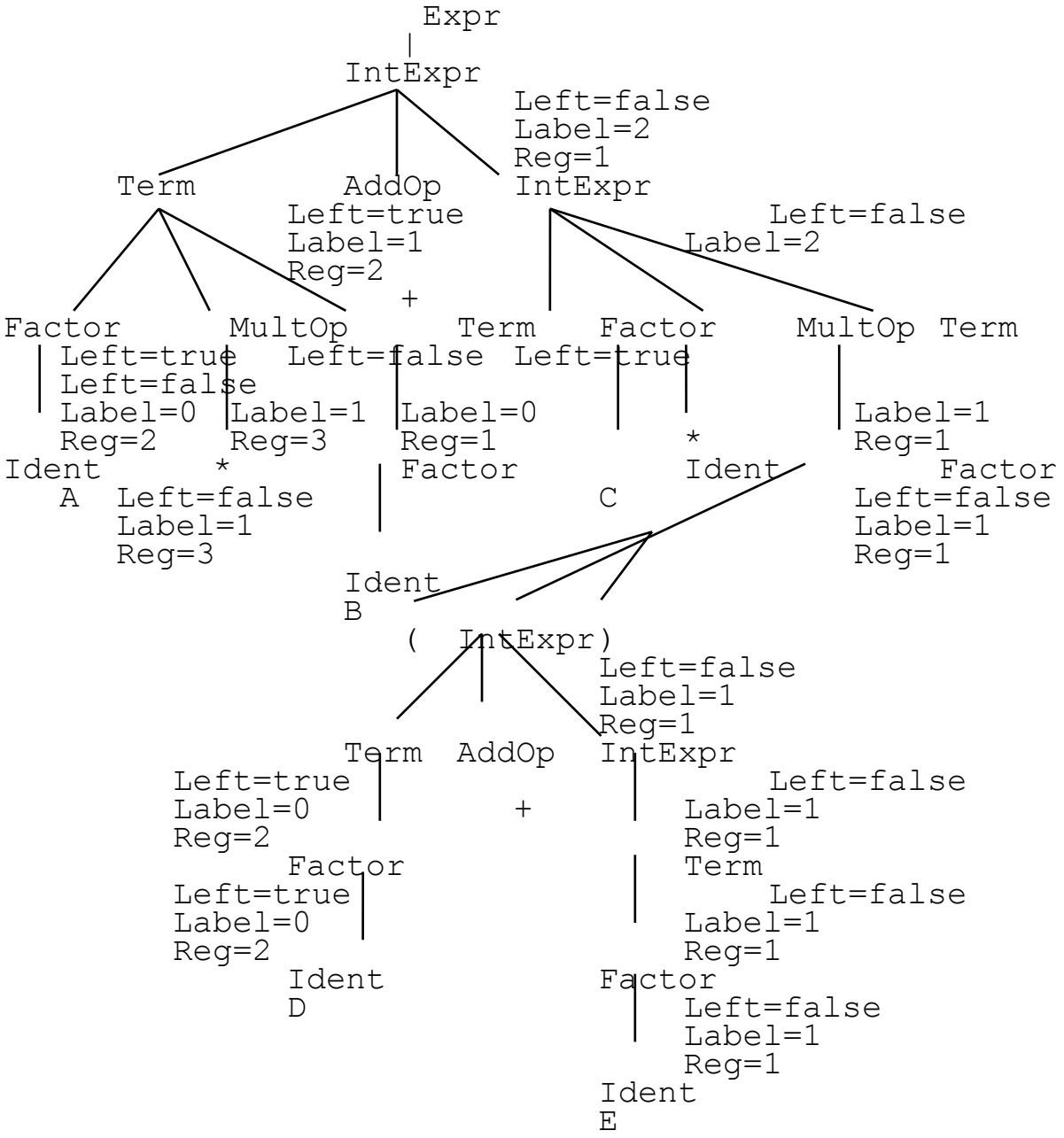


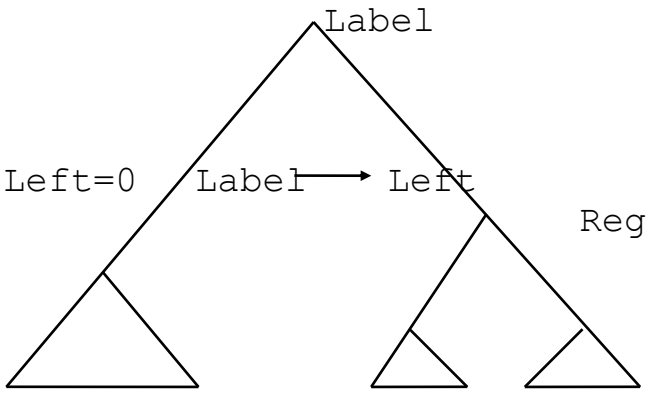
Рис. 8.18.

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 [9].

Пусть мы произвели разметку дерева разбора так же, как и в предыдущем алгоритме. Назначение регистров будем производить в соответствии со схемой рис. 8.19.

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.




```
Reg<0>= ((Left<0>==Label<0>)
    &&(Left<0>!=0))
    ? Label<0>+1
    : Label<0>
```

Рис. 8.19.

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>

: //Label<3>==Label<1>
                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>
        : "LOAD" + Val<1> + "R1".

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:

```
LOAD E,R1 - загрузить E на 1 регистр
ADD D,R1 - сложить D и E и результат заслать в 1
          регистр
MUL C,R1 - умножить C на D+E с результатом в 1
          регистре
MOVE R1,R2 - переслать результат в регистр R2
LOAD B,R1- загрузить B в 1 регистр
```

```
MUL A,R1 - умножить A на B с результатом в 1 регистре
ADD R1,R2 - сложить A*B и C*(D+E) и результат заслать
          во регистр
```

В приведенных атрибутных схемах предполагалось, что регистров достаточно, чтобы правильно транслировать любое выражение. Если это не так, приходится усложнять схему трансляции и при необходимости сбрасывать содержимое регистров в память (или магазин).

8.7. 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, пользуясь очевидными правилами:

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 "вычисления 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>=NodeLable<0>+":"+Code<1>+Code<3>.

RULE
BoolExpr ::= BoolExpr 'OR' BoolExpr
```

```
SEMANTICS
TrueLab<1>=TrueLab<0>; FalseLab<1>=NodeLab<3>;
FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>;
Code<0>=NodeLable<0>+":"+Code<1>+Code<3>.

RULE
BoolExpr ::= F
SEMANTICS
Code<0>="GOTO FalseLab<0>".

RULE
BoolExpr ::= T
SEMANTICS
Code<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. 8.20.

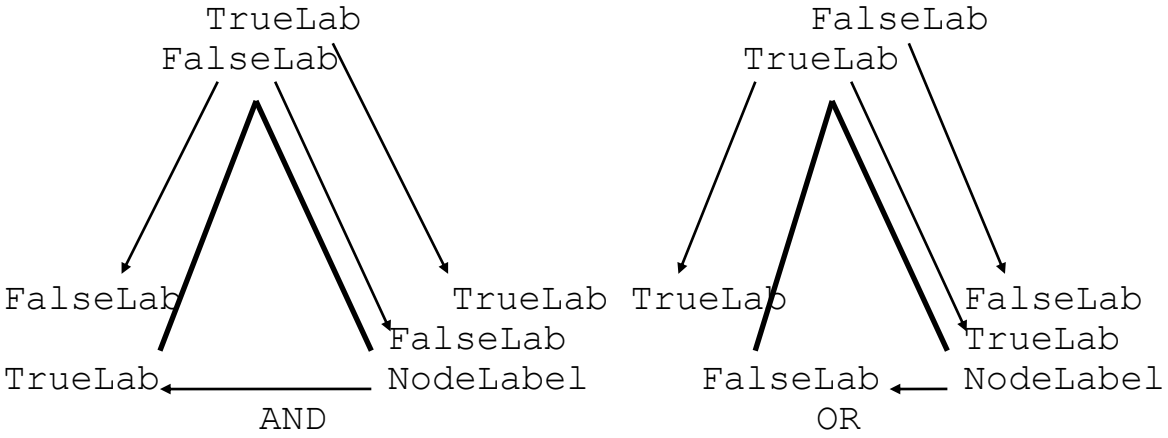


Рис. 8.20.

Сопоставим каждому атрибутированному дереву в этой атрибутной грамматике текст (трансляцию), полученный следующим образом в результате обхода дерева сверху вниз слева направо: при входе в вершину будем генерировать ее номер, в вершине F будем генерировать 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 и трансляцию, изображенные на рис. 8.21 и 8.22.:

F OR (F AND T AND T) OR T

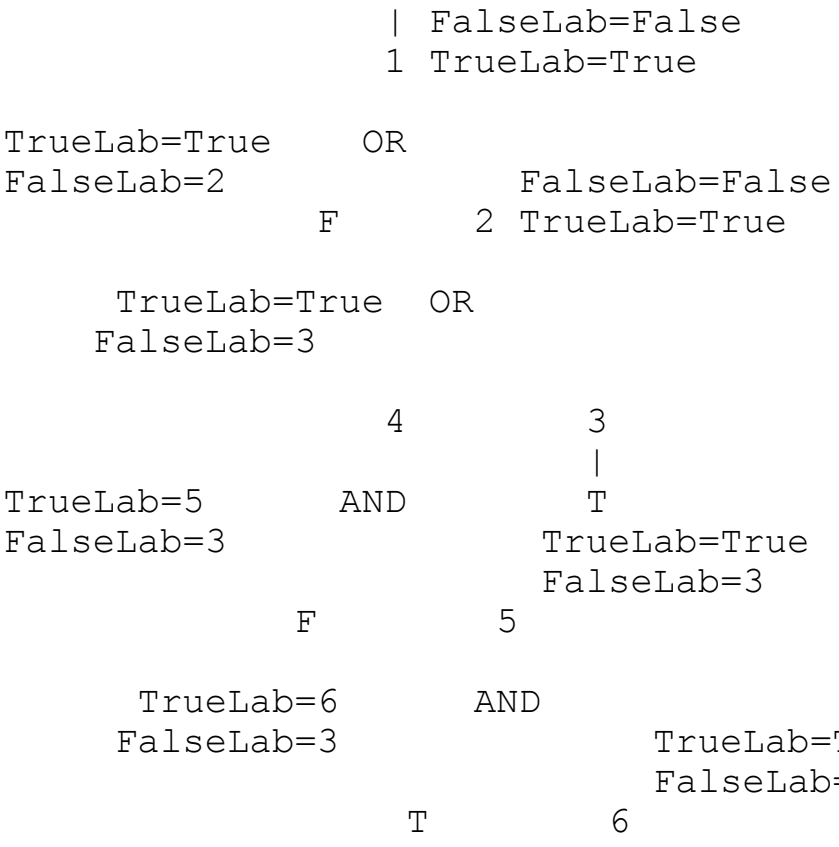


Рис. 8.21

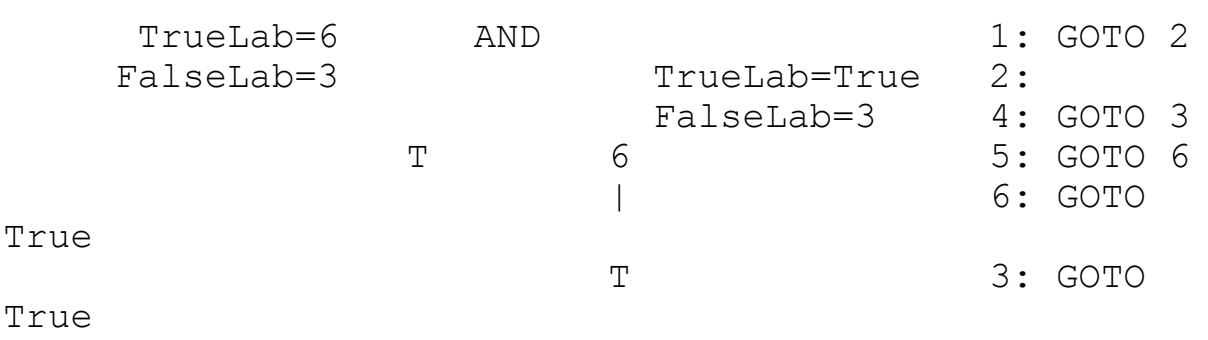


Рис. 8.22

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, либо пусто. 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 1. 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.

Это утверждение является частным случаем следующего.

Утверждение 2. 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 можно провести индукцией по высоте дерева. Для деревьев высоты 1, соответствующих правилам BoolExpr ::= F и BoolExpr ::= T, утверждение очевидно из правил грамматики. Пусть дерево имеет высоту n>1. Зависимость атрибутов для дизъюнкции и конъюнкции приведена на рис. 8.23

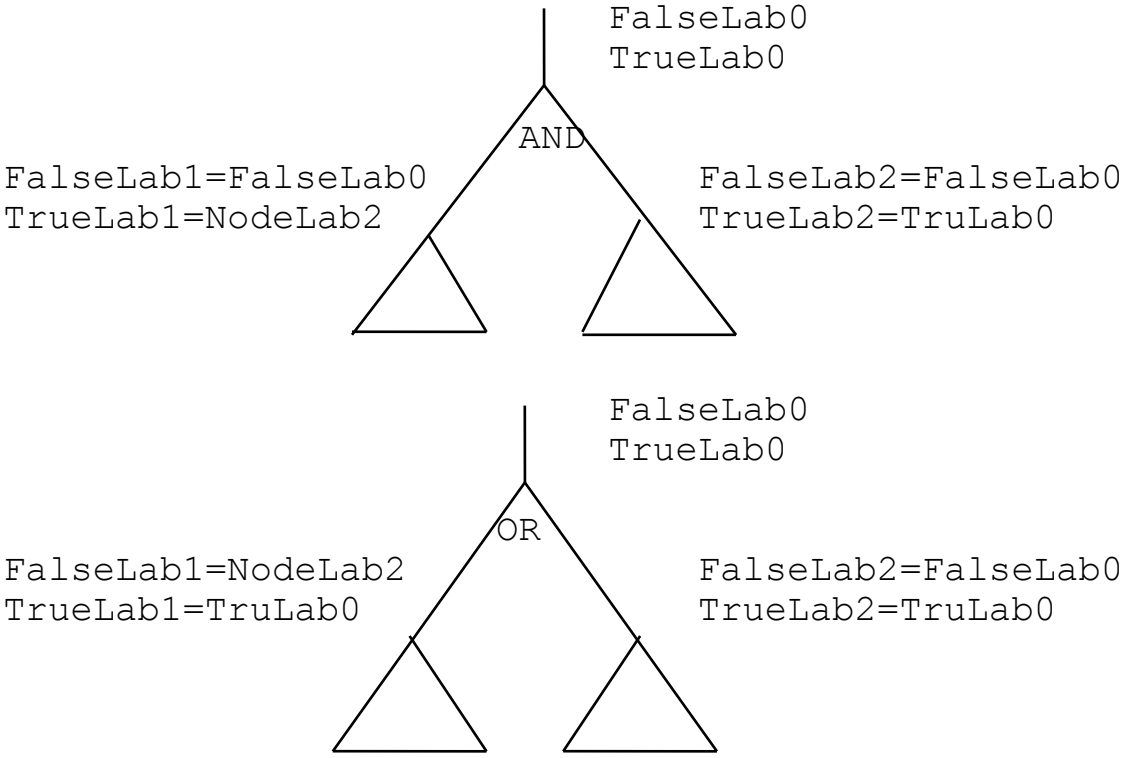


Рис. 8.23

Если для конъюнкции значение левого поддерева ложно и по индукции вычисление левого поддерева завершается командой GOTO FalseLab1, то и интерпретация всего дерева завершается командой GOTO FalseLab0 (=FalseLab1). Если значение левого поддерева истинно, то его интерпретация завершается командой GOTO TrueLab1 (=NodeLab2). Если значение правого поддерева ложно, то интерпретация всего дерева завершается командой GOTO FalseLab0 (=FalseLab2). Если же оно истинно, интерпретация всего дерева завершается командой GOTO TrueLab0 (=TrueLab2). Аналогично - для дизъюнкции.

Доказательство утверждения 1 следует из того, что метками вершины дерева логического выражения являются TrueLab=True и FalseLab=False.

Добавим теперь новое правило в предыдущую грамматику:

```
RULE
BoolExpr ::= Ident
SEMANTICS
Code<0>=”if (Val<1>==T)GOTO TrueLab<0>
        else GOTO FalseLab<0>”;
```

Тогда, например, для предыдущего выражения получим следующую программу:

```
1: if (F==T) then GOTO True else GOTO 2
2:
4: if (F==T) GOTO 5 else GOTO 3
5: if (T==T) GOTO 6 else GOTO 3
6: if (T==T) GOTO True else GOTO 3
3: if (T==T) GOTO True else GOTO False
```

При каждом конкретном наборе данных эта программа превращается в программу вычисления логического значения.

Утверждение 3. В каждой строке программы, сформированной предыдущей атрибутной схемой, одна из меток совпадает с меткой следующей строки.

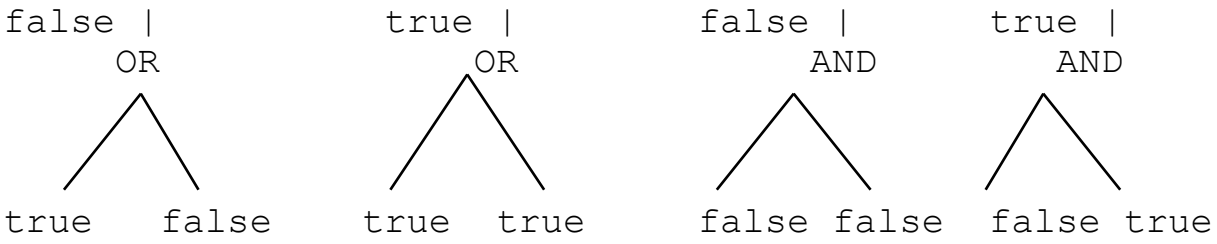
Действительно, по правилам наследования атрибутов TrueLab и FalseLab, в правилах для дизъюнкции и конъюнкции либо FalseLab, либо TrueLab принимает значение метки следующего поддерева. Кроме того, как значение FalseLab, так и значение TrueLab, передаются в правое поддерево от предка. Таким образом, самый правый потомок всегда имеет одну из меток TrueLab или FalseLab, равную метке правого брата соответствующего поддерева. Учитывая порядок генерации команд, получаем утверждение.

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>=NodeLable<0>+":"+Code<1>+Code<3>.

RULE
BoolExpr ::= BoolExpr 'OR' BoolExpr
SEMANTICS
TrueLab<1>=TrueLab<0>; FalseLab<1>=NodeLab<3>;
FalseLab<3>=FalseLab<0>; TrueLab<3>=TrueLab<0>;
Sign<1>=true; Sign<3>=Sign<0>;
Code<0>=NodeLable<0>+":"+Code<1>+Code<3>.

RULE
BoolExpr ::= NOT BoolExpr
SEMANTICS
FalseLab<1>=TrueLab<0>; TrueLab<1>=FalseLab<0>;
Sign<1>=! Sign<0>;
Code<0>=Code<1>.

RULE
BoolExpr ::= F
SEMANTICS
Code<0>="GOTO FalseLab<0>".

RULE
BoolExpr ::= T
SEMANTICS
Code<0>="GOTO TrueLab<0>".

RULE
BoolExpr ::= Ident
SEMANTICS
Code<0>="if (Val<1>==T) GOTO TrueLab<0>
        else GOTO FalseLab<0>".
```

Правила наследования атрибута Sign приведены на рис. 8.24.



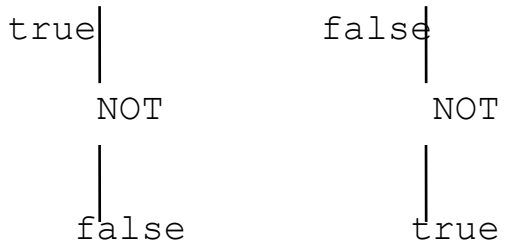


Рис. 8.24

Программу желательно сформировать таким образом, чтобы else-метка была как раз меткой следующей вершины. Как это можно сделать, следует из утверждения 4.

Утверждение 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.

Эти два утверждения позволяют заменить последнее правило атрибутной грамматики следующим образом:

```
RULE
BoolExpr ::= Ident
SEMANTICS
Code<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
<TST Ident>;
Code<0>=Sign<0>
         ? ”BNE TrueLab<0>”
         : ”BEQ FalseLab<0>”.
```

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.

Приведем несколько примеров. Выражение A AND (B OR C) транслируется в последовательность команд рис. 8.25. Выражение (NOT((A==B)OR(!=D)))AND(NOT((E<F)AND(G>H))) транслируется в последовательность команд рис. 8.26.

```
        TST A
        BEQ False
        TST B
        BNE True
        TST C
        BEQ False
 True:
False:. . .
```

Рис. 8.25

```
        CMP A,B
        BEQ False
        CMP C,D
        BNE False
        CMP E,F
        BGE False
        CMP G,H
        BGT False
 True:
False:
```

Рис. 8.26

8.8. Выделение общих подвыражений

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 номера всех переменных устанавливаются равными 0. При каждом присваивании переменной ее номер 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 (рис. 8.27).

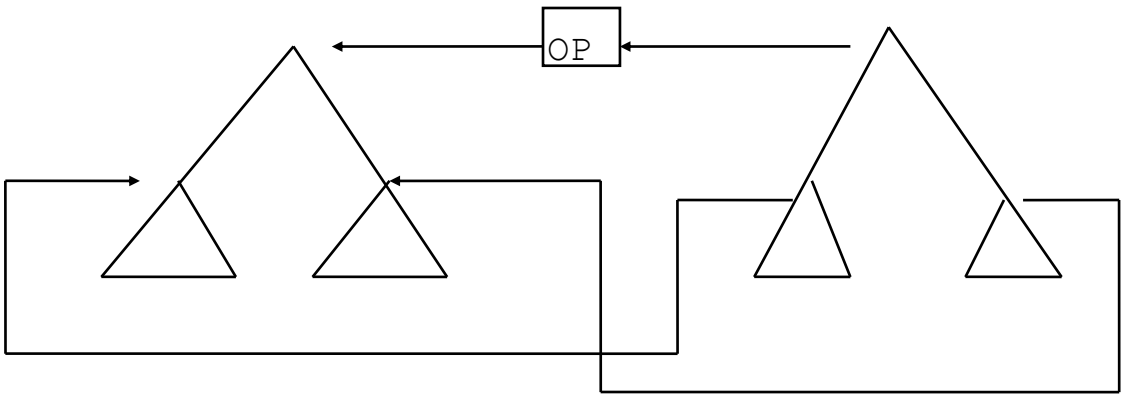


Рис.8.27

Рассмотрим теперь реализацию алгоритма выделения общих подвыражений. Поддерживаются следующие глобальные переменные:

Table - таблица переменных; для каждой переменной хранится ее номер (Count) и указатель на вершину дерева выражений, в которой переменная встретилась в последний раз в правой части (Last);

OpTable - таблица списков общих подвыражений, связанных с каждой операцией (Addr - указатель на вершину дерева, List - продолжение списка);

С каждой вершиной дерева выражения связана запись (Тип LisType):

```
struct NodeType {
   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;
}
```

Все общие подвыражения собраны в список (с типом элемента LisType), начинающийся с OpTable[Op], как это изображено на рис. 8.28.

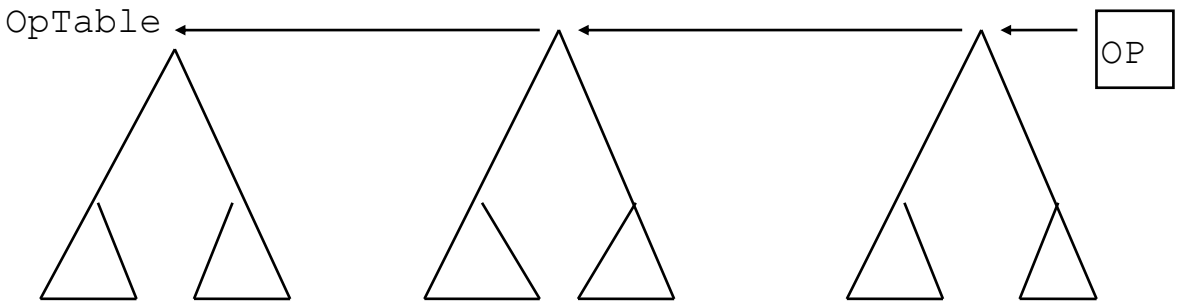


Рис. 8.28

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->Node<0>->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
ListType * 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=new 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: то ли метить предыдущую вершину для сброса, то ли вычислять полностью поддерево.

8.9. Генерация оптимального кода методами синтаксического анализа

8.9.1. Сопоставление образцов

Техника генерации кода, рассмотренная выше, основывалась на однозначном соответствии структуры промежуточного представления и описывающей это представление грамматики. 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.

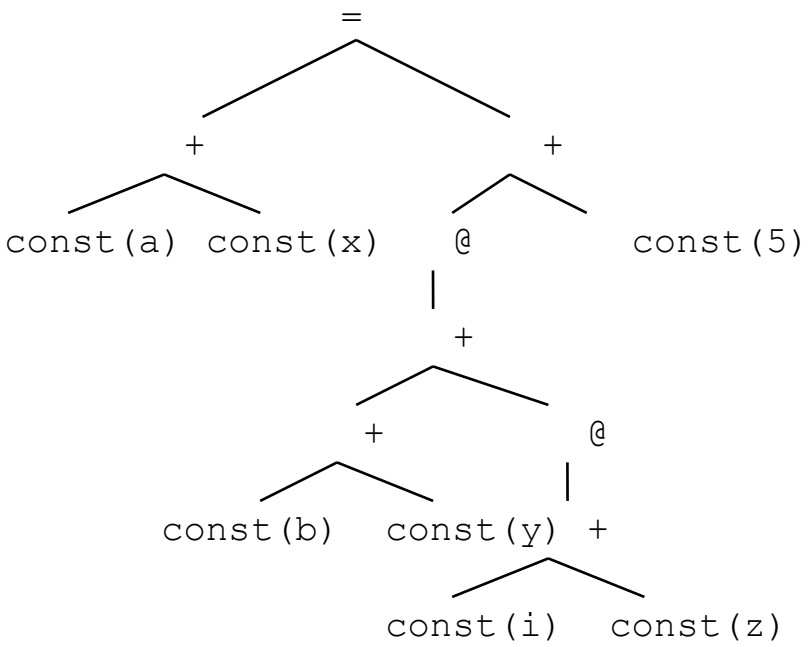


Рис. 8.29

На рис. 8.29 показано промежуточное дерево для оператора a=b[i]+5, где a,b,i - локальные переменные, хранимые со смещениями x,y,z 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 оператора.

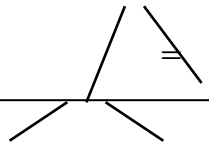
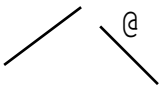
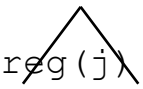
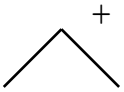
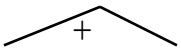
Но-мер	Образец	Машинная команда Правило грамматики	Стои-мость
1	const(c)	MOV #c,Ri Reg->Const	2
2	= 	MOVE Rj,c(Ri)	4
	+ reg(j) reg(i) const(c)	Stat->'=' '+' Reg Const Reg	
3	@ 	MOVE c(Rj),Ri	4
	+		
	reg(j) const(c)	Contents -> '@' '+' Reg Const	
4	+ 	ADD #c,Ri	3
	reg(i) const(c)	Reg -> '+' Reg Const	
5	+ 	ADD Rj,Ri	2
	reg(i) reg(j)	Reg -> '+' Reg Reg	
6	+ 	ADD c(Rj),Ri	4
	reg(i) @ | +	Reg -> '+' Reg '@' '+' Reg Const	
	reg(j) const(c)		
7	@	MOVE (R),R	2
	| Reg	Reg -> Contents	

Рис. 8.30

На рис.8.30 показан пример сопоставления образцов машинным

командам. Приведены два варианта задания образца: в виде дерева и в виде правила контекстно-свободной грамматики. Для каждого образца указана машинная команда, реализующая этот образец, и стоимость этой команды. Стоимость может определяться различными способами, и здесь мы не рассматриваем этого вопроса. На рис. 8.31 приведен пример покрытия промежуточного дерева рис. 8.29 образцами рис. 8.30. В рамки заключены фрагменты дерева, сопоставленные образцу правила, номер которого указывается в левом верхнем углу рамки. В квадратных скобках указаны результирующие вершины.

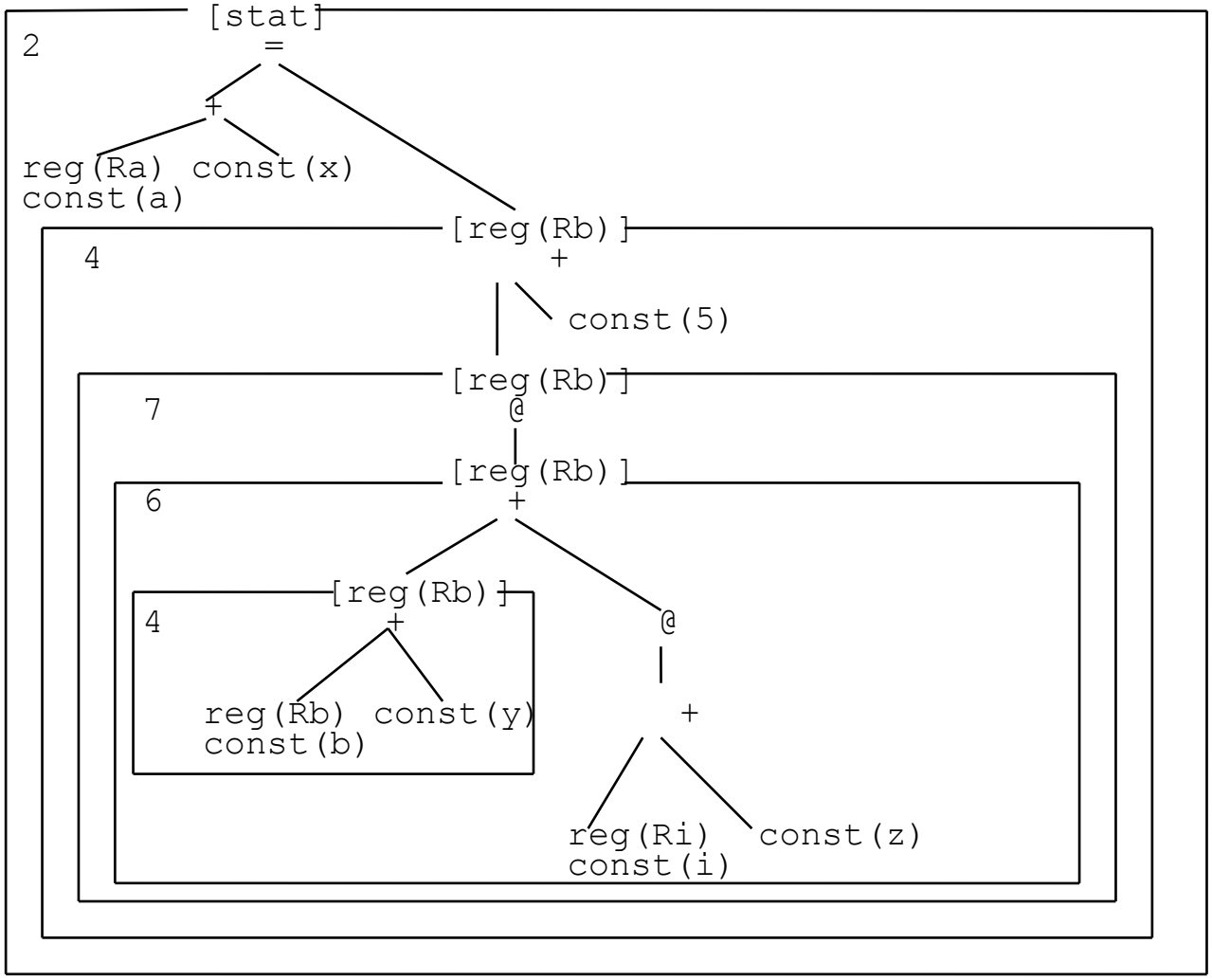


Рис. 8.31

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)
```

Как правило, одни и те же конструкции исходной (или промежуточной) программы можно реализовать различными последовательностями машинных команд. Это соответствует тому, что имеются различные покрытия промежуточного представления. 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 [10,11]. Мы здесь рассмотрим алгоритм [12], комбинирующий возможности синтаксического анализа и динамического программирования, в основу которого положен синтаксический анализ неоднозначных грамматик (модифицированный алгоритм Кока, Янгера и Касами [13,14]) более эффективный в реальных приложениях. Этот же метод может быть применен и тогда, когда в качестве промежуточного представления используется дерево.

8.9.2. Синтаксический анализ для Т-грамматик

Обычно код генерируется из некоторого промежуточного языка с довольно жесткой структурой. В частности, для каждой операции известна ее размерность (число операндов). Назовем грамматики, удовлетворяющие этим ограничениям, *Т-грамматиками*.

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 оценкой. 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 $a_1...a_n$ 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 $a_i...a_k$ входной цепочки $a_1...a_n$, проходя слева-направо по

$a_i...a_k$. 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 8.1. Тип Titem в алгоритме 8.1 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 8.1:
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];
// Распознование входной цепочки
 for (int i=n-1;i>=0;i--)
 {for (для каждого правила A -> a[i] y из P)
   {//считаем,что l[i]=0 для символов-не знаков
    //операций
     int j;
     if (l[i]>0)
         {j=i+1;


         h=[A->a[i].y];
         do //Сопоставимы ли a[i]y и a[i]..a[i+l[i]-1]
          {Пусть h==[A->u.Xv]
          if   (X в T)
            if (X==a[j]) j=j+1; else break;
          else // X в N
            if  (X->w в r[j]) j=j+l[j];
           else break;
          h=[A->uX.v];
         }// Перейти к следующему символу
```

```
        while( j!=i+l[i]);
         r[i]=r[i]+{(A->a[i]y)};
    }//for
    // Проверить 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)};
  Проверить, принадлежит ли (S->w) множеству r[0];
```

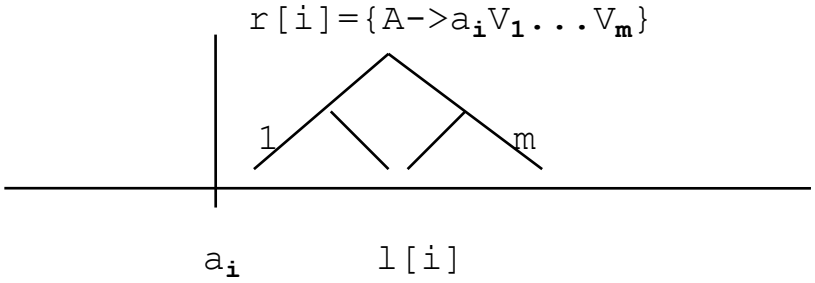


Рис. 8.32

Работа алгоритма иллюстрируется рис. 8.32. Множества r[i] имеют размер O(|P|). Очевидно, что алгоритм имеет временную и емкостную сложность O(n).

Рассмотрим вновь пример рис. 8.29. В префиксной записи приведенный фрагмент программы записывается следующим образом:

= + a x + @ + + b y @ + i z 5

На рис. 8.33 приведен результат работы алгоритма. Правила вычисления стоимости приведены в разделе 8.9.3.

Операция	Длина	Правила (стоимость	
=	14	2(22)	
+	2	4(5)	5(6)
a	0	1(2)	
x	0	1(2)	
+	9	4(16)	5(17)
@	8	7(13)	
+	7	5(15)	6(11)
+	2	4(5)	5(6)
b	0	1(2)	
y	0	1(2)	
@	3	3(6)	7(7)
+	2	4(5)	5(6)
i	0	1(2)	
z	0	1(2)	
5	0	1(2)	

Рис. 8.33

Пусть G - это T-грамматика. Для каждой цепочки z из L(G) можно построить дерево выражения. Мы можем переписать алгоритм так, чтобы он работал с деревьями выражений, а не с префиксными выражениями. 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, чтобы иметь соответствие с порядком вычисления в алгоритме 8.1. 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 8.1.

```
Алгоритм 8.2:
 Tnode * root;
  bool MATCHED(Tnode * n, Titem h)
  {bool matching;
   int m=Arity(X);
   пусть h==[A->u.Xvy], v== v1 v2 ... vm,
   if (X in T)// сопоставление правила
     if (m==0) //if l[i]==0
        if (X==n->op)  //if X==a[j]
         return(true);
        else return(false);
     else // if l[i]>0
      if (X==n->op) //if X==a[j]
        {matching=true;
         for (i=0;i<m;i++)   //j=j+l[j]
         matching=matching && //until j=i+l[i]
                  MATCHED(n->son[i],[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);
  }//match

void PARSE(Tnode * n);
 {//PARSE
   for (i=Arity(n->op)-1;i>=0;i--)
   //for (i=n-1; i>=0;i--)
     PARSE(n->son[i]);
   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)};
 }//PARSE
  //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. Если Т-грамматика неоднозначная, то можно построить несколько различных выводов для заданной входной цепочки.

Алгоритмы 8.1 и 8.2 "универсальные" в том смысле, что конкретные грамматики выражений и образцов являются, по-существу, параметрами этих алгоритмов. Для каждой конкретной грамматики можно написать свой алгоритм поиска образцов. Например, в случае нашей грамматики выражений и приведенных на рис. 8.30 образцов алгоритм 8.2 может быть представлен атрибутной грамматикой, приведенной ниже. Для каждого правила имеется два типа образцов: "внутренние", соответствующие правилам-образцам, и "внешние", приходящие сверху через атрибут Match предка. Атрибут Match представлен как вектор образцов вида <$Pattern_1$, ... $Pattern_n$>. Каждый из образцов имеет вид либо <op op-list>, где op - операция в данной вершине, а op-list - список ее операндов, либо образец - это нетерминал N. В первом случае op-list "распределяется" по потомкам вершины для дальнейшего сопоставления, во втором сопоставление считается успешным, если есть правило N->w, где w - состоит из образцов, успешно сопоставленных потомкам данной вершины. Помимо атрибута Match, образцы могут задаваться и в соответствии с правилами, возможно применимыми в данной вершине. Эти два множества образцов могут пересекаться. Синтезируемый атрибут Pattern (также вектор) дает результат сопоставления по вектору-образцу Match.

```
RULE
Stat ::= '=' Expr Expr
```

/*Этому правилу соответствует один образец 2. Поэтому в качестве образцов потомков через их атрибуты Match передаются, соответственно, <'+' Reg Const> и <Reg>.*/

```
SEMANTICS
Match<2>=<'+' Reg Const>:
Match<3>=<Reg>;
Pattern<0>[1]=Pattern<2>[1]&Pattern<3>[1].

RULE
Expr ::= '+' Expr Expr
```

/*Образцы, соответствующие этому правилу, следующие:

(4) Reg -> '+' Reg Const

(5) Reg -> '+' Reg Reg

(6) Reg -> '+' 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.*/

```
SEMANTICS
Match<2>=<Reg,Reg,Reg>;
Match<3>=<Const,Reg,<'@' '+' Reg Const>>;
if (Match<0> содержит образец
 <'+' Reg Const> в i-й позиции)
Pattern<0>[i]=Pattern<2>[1]&Pattern<3>[1];

if (Match[0] содержит Reg в j-й позиции)
Pattern<0>[j]=(Pattern<2>[1]&Pattern<3>[1])
              |(Pattern<2>[2]&Pattern<3>[2])
              |(Pattern<2>[3]&Pattern<3>[3]);
Остальным значениям Pattern<0> присвоить false.

RULE
Expr ::= '@' Expr
```

/*Образцы, соответствующие этому правилу, следующие:

(3) Reg -> '@' '+' Reg Const

(7) Reg -> '@' 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 сопоставлении образцов предков левой части данного правила атрибуту Match могут быть переданы образцы <'@' '+' Reg Const> (из образца 6) и Reg.*/

```
SEMANTICS
Match<2>=<<'+' Reg Const>,Reg>;
if (Match<0> содержит <'+' Reg Const> в i-й позиции)
Pattern<0>[i]=Pattern<2>[1];
if (Match<0> содержит Reg в j-й позиции)
Pattern<0>[j]=Pattern<2>[1]|Pattern<2>[2];
Остальным значениям Pattern<0> присвоить false.
```

```
RULE
Expr ::= Const
SEMANTICS
if (Pattern<0> содержит Const в j-й позиции)
Pattern<0>[j]=true;
Остальным значениям Pattern<0> присвоить false.
```

Для дерева рис. 8.29 получим значения атрибутов, приведенные на рис. 8.34. Здесь M обозначает Match, P - Pattern, C - Const, R - Reg.

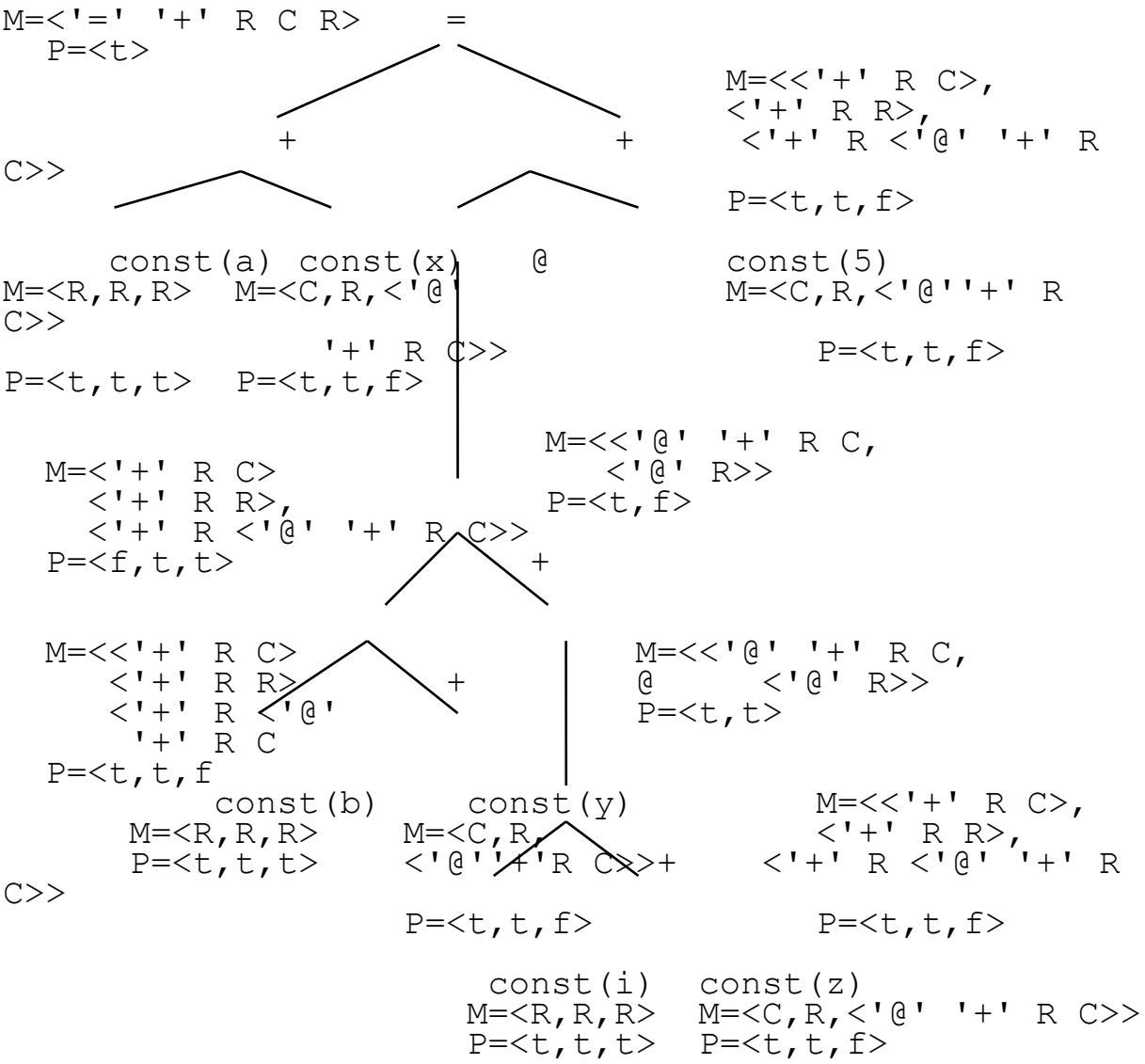


Рис. 8.34

8.9.3. Выбор дерева вывода наименьшей стоимости

Т-грамматики, описывающие системы комад, 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 $p:A::=z_0\ X_0\ z_1...X_k\ z_k$, где z_i из T* для $0<=i<=k$ и X_j из N для $1<=j<=k$. Вершина n имеет потомков $n_1,...,n_k$, которые соответствуют нетерминалам $X_1,...,X_k$. Значения атрибутов стоимости вычисляются в порядке снизу вверх. Вначале атрибуты стоимости инициализируются неопределенным значением UndefinedValue. Предположим, что значения атрибутов стоимости для всех потомков $n_1,...,n_k$ вершины n вычислены. Если формула

```
A.a=f(Xi.b,Xj.c,...) для 1<=i,j<=k
```

сопоставлена правилу p, то выполняется формула

```
n.a=f(ni.b,nj.c,...),
```

вычисляющая для правила p значение 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 8.2 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>=0;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);
     }
   }
}
```

Когда выбрано дерево вывода "наименьшей" стоимости, вычисляются значения атрибутов, сопоставленных вершинам дерева вывода, и генерируются соответствующие машиннные 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 $p:A::=z_0\ X_0\ z_1...X_k\ z_k$, связанное с указанной вершиной n, и заданный нетерминал A, чтобы определить соответствующие им вершины $n_1,...,n_k$ и нетерминалы $X_1,...,X_k$. Затем вычислитель рекурсивно обходит каждую вершину n_i, имея на входе нетерминал X_i.

Литература

1. Aho A., Sethi R., Ullman J. Compilers: principles, techniques and tools. N.Y.: Addison-Wesley, 1986.
2. Грис Д. Построения компиляторов для цифровых вычислительных машин. М.: Мир, 1975.
3. Fraser C.W., Hanson D.R. A Retargetable compiler for ANSI C.// SIGPLAN Notices. 1991. V 26.
4. Надежин Д.Ю., В.А.Серебряков, В.М.Ходукин. Промежуточный язык Лидер (предварительное сообщение)//Обработка символьной информации.М.: ВЦ АН СССР, 1987. С. 50-63.
5. Ахо А., Ульман Д. Теория синтаксического анализа, перевода и компиляции. М.: Мир, 1978.
6. Кнут Д. Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы. М.: Мир, 1976.
7. Лавров С.С., Гончарова Л.И. Автоматическая обработка данных. Хранение информации в памяти ЭВМ. М.: Наука, 1971.
8. Адельсон-Вельский Г.М., Ландис Е.М. Один алгоритм организации информации// ДАН СССР. 1962. Т. 146. N 2. С. 263-266.
9. 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.
10. Emmelman H.,Schroer F.W.,Landweher R. BEG-a generator for efficient back-ends// ACM SIGPLAN. 1989. V.11. N 4.p.227-237
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. A. Bezdushny, V. Serebriakov. 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. Graham S.L., Harrison M.A., Ruzzo W.L. Am improved context-free recognizer// ACM Trans. Program. Languages and Systems. 1980. N.2.
14. Harrison M.A. Introduction to formal language theory. Reading, Mass.: Addison-Wesley, 1978.