TRANSPORTA UN SAKARU INSTITŪTS

ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРОВ

Конспект лекций

Составитель О. Б. Люсин

Riga 2012

004
П791

Transporta un sakaru institūts
Институт транспорта и связи

П791 Проектирование компиляторов: Конспект лекций. Сост. О. Б. Люсин. Рига: Институт Транспорта и связи, 2012. 176 с.

Эта книга представляет собой учебное пособие, основанное на курсе лекций, читаемом автором в рамках дисциплины «Проектирование компиляторов» и предназначена для студентов компьютерных специальностей. Оно включает также материалы для лабораторных работ и контрольные вопросы по ключевым излагаемым темам.

Содержание пособия включает в себя «классические» разделы предмета, как-то лексический и синтаксический анализ, а также построение внутренних форм представления исходной программы, различных видов оптимизации, организации памяти и генерации кода.

Пособие может быть полезно всем, кто интересуется вопросами проектирования компиляторов (интерпретаторов) для современных языков программирования.

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ В АВТОМАТИЗАЦИЮ ПРОГРАММИРОВАНИЯ

Под автоматизацией программирования понимается создание методов и способов облегчения или полного отстранения человека от написания программ для вычислительной системы. Чаще всего в роли таких инструментов выступают языки программирования различного уровня, однако существуют методики использования для автоматизации программирования различных средств ИИ и других подходов, но они не являются предметом этого курса.

1.1. Основная проблема автоматизации

Все языки программирования по степени абстракции от аппаратной реализации вычислительной системы можно разделить на следующие типы:

- машинный язык – система инструкций (команд) процессора, интерпретируется непосредственно процессором;
- язык среднего уровня – компилируемый в машинные инструкции символический язык (например, ассемблер, автокод), имеет непосредственный доступ к аппаратным средствам вычислительной машины;
- *я*зык высокого уровня (HLL – High Level Language) – характеризуется высоким уровнем абстракции и полной машинно-независимостью. Все современные языки программирования относятся к этому типу языков. Их часто также называют алгоритмическими языками.

Можно классифицировать языки программирования и по другим критериям.

По сфере применения:

– универсальные языки – ALGOL60/68, PL/1, C/C++, JAVA…;
– проблемно-ориентированные языки – FORTRAN, COBOL, PROLOG, ADA;
– специализированные – Ляпас, Аналитик, MATLAB…;

По парадигме программирования:
– процедурные языки – C, FORTRAN;
– объектно-ориентированные – Smalltalk, Scala, JAVA, Objective C;
– функциональные – Scheme, Haskell;
– мультипарадигмные – C++, Python;
– другие языки – Lisp, Prolog.

По принципу конструирования:
– синтаксически-ориентированные языки;
– подпрограммно-ориентированные языки.

По характеру перевода:
– компилируемый – C, C++;
– интерпретируемый – JAVA, Python, Haskell.

По виду типизации:
– статическая – C, C++, Haskell;
– динамическая – Python, Ruby.

Таким образом, автоматизация программирования проявляет себя как в создании самих языков программирования, так и в построении системных программ-переводчиков для исходных пользовательских программ, написанных на конкретном языке высокого уровня – в ассемблерный и далее – машинный коды.

Типы программ-переводчиков:
компиляторы;
трансляторы (ассемблеры);
интерпретаторы.

Разработка первого компилятора для Фортрана заняла 18 человеколет.

1.2. Основные элементы системы перевода

Человеческий перевод начинается с ознакомления (анализа) с исходным текстом.

Машинный (программный) перевод начинается с декомпозиции исходного текста.

Основные шаги программного перевода:
1. Декомпозиция исходного текста.
2. Анализ составляющих языка.
3. Синтез целевого текста программы.

В основе любого языка лежит определенный алфавит.

В состав алфавита языка программирования входят:

- буквы – A, a...
- цифры – 1, 2, 3...
- специальные символы – +, -, (,),...
- управляющие (невидимые) символы – Enter, TAB...

Из алфавита создаются более крупные составляющие (объекты), которые, что неудивительно, называются словами. Например:

- BEGIN
- 1000
- *
- и т.д.

Слова бывают простые или составные, например, слово „16.45” состоит из трех слов.

Словарный состав языка, представляющий собой множество всех допустимых слов для данного алфавита, называется *лексиконом*.

Правила образования правильных слов из символов алфавита называются *лексическими правилами*, или просто *лексикой* языка.

Из слов языка строятся языковые конструкции, называемые предложениями. Построение предложения в данном языке строго следует *синтаксису* этого языка – то есть правилам построения предложений. *Синтаксические* правила часто называют грамматическими правилами, или грамматикой. Смысловое содержание предложения – то есть его суть – называется *семантикой*. Машинная семантика есть правильная последовательность машинных команд, исполнение которых в определенной последовательности приведет к желаемому результату.

Процесс перевода из одного языка в другой можно представить в виде следующих этапов:

1) замена символов алфавита одного языка на символы из алфавита другого языка;

2) замена лексики одного языка на лексику другого языка;

3) замена синтаксиса одного языка на синтаксис другого языка.

В процессе перевода семантика обязана быть неизменной. То есть семантика выходного текста равна семантике исходного текста..

2. ФОРМАЛЬНЫЕ СИСТЕМЫ

Любая формальная система является неинтерпретируемой (необъяснимой) системой обозначений и понятий, состоящей из:

1) алфавита;

2) множества слов (аксиом);

3) конечного множества отношений элементов формальной системы, называемых правилами вывода.

В качестве примеров формальных систем можно привести, например, естественные языки, с помощью которых общаются люди, язык арифметики и алгебры (булева, матричная и т.д.). Языки программирования тоже являются формальными системами.

Построение формальной системы – это создание и применение специальной системы знаков, терминов, понятий, правил и формул для описания ранее неизвестных фактов, процессов, явлений или взаимоотношений между ними, структуры языка, правил формирования его компонентов, а также верификации истинности построения того или иного предложения языка. Такая система, используемая для построения формальной системы, также является формальной.

Конечным и самым важным этапом построения формальной системы является верификация – или строгое доказательство правильности той или иной конструкции языка. Доказательство может быть либо генерационным, либо аналитическим.

Таким образом, языки программирования можно классифицировать и по этому признаку:

- генерационно-определяемые языки;
- аналитически-определяемые языки.

2.1. Основные определения

Ниже приведены некоторые основные определения формальной системы, применяемой для анализа и конструирования языков программирования, необходимые при дальнейшем изучении курса. Практически все определения используют понятие «множество», которое есть совокупность

некоторых объектов, объединяемых одним или несколькими общими признаками, и записывается, например, {1,2,3,4,5}.

Пустое множество обозначается как {}.

Определение 1

Язык, который предназначается для изучения или исследования, является объектным языком.

Определение 2

Язык, который применяется для изучения объектного языка, называется метаязыком.

Определение 3

A – алфавит некоторого языка L есть непустое конечное множество объектов, называемых символами алфавита.

Определение 4

Символом называется некоторый графический объект, воспринимаемый как единое целое и несущий соответствующую смысловую нагрузку. В языках программирования символ может быть составным.

Определение 5

Строка (цепочка, chain, string) представляет собой конечную последовательность некоторых разрешенных в данном алфавите символов. Порядок следования символов в строке существенно важен.

Пример строки – "ABC", "Вася", "12345".

Алфавит изображается в виде множества входящих в него одиночных символов: A={a,b,c}.

Строка, не содержащая ни одного символа алфавита, называется пустой строкой и обозначается как ε – эпсилон.

Определение 6

Алфавит T есть конечное множество символов объектного языка, называемых терминальными символами.

Определение 7

Алфавит N есть конечное множество символов мета-языка, называемых нетерминальными символами.

Следствие определения 7:

в языке, определенном на алфавите V = N + T, строки содержат как терминальные, так и нетерминальные символы.

Определение 8

Процесс (операция) по формированию строки из элементов алфавита называется конкатенацией (или сцеплением). Конкатенация определяется как присоединение справа некоторого одиночного символа (или нескольких) к предшествующему символу в строке.

Например, процесс создания конспекта лекций есть конкатенация.

Если P и Q – два множества строк, то операция конкатенации этих множеств записывается в следующем виде:

PQ или P*Q (PQ)

Например: P={10}, Q={1,00}.

Тогда PQ={101,1000}.

Определение 9

Для любого конечного множества символов алфавита A запись вида A* (A в «степени» *) обозначает множество всех строк, которые возможно получить из символов этого алфавита. При этом определяется, что A* всегда содержит пустую строку.

Операция «*» называется итерацией, или звездой Клини (Kleene star).

$A^* = A^0 + A^1 + A^2 + A^3 + A^4 + ... + A^n + ...$

Следствие определений 8 и 9:

конкатенация множества с самим собой обозначается как PP или P^2.

Аналогично:

P^i = P*P*…….P, где i – степень конкатенации.

При этом: $P^0 = \{\varepsilon\}$.

Запись вида $A^+A^+ = A^1 + ... A^n + ...$ не содержит пустой строки, этим определяется различие применения символов - * и + в теории формальных языков.

2.2. Формальное определение языка и грамматики

Язык L есть подмножество множества T*.

Правило – есть упорядоченная пара (U,x), которая записывается как U::=x, где U есть нетерминальный символ, U

$\in$ N, а x $\in$ (N+T)*, где x – строка. В некоторых источниках правило называют продукцией.

Нетерминальный символ Z $\in$ N называется начальным символом грамматики и должен появиться в левой части по крайней мере одного правила грамматики.

Грамматика G (N, T, Z, P) представляет собой формальную систему в виде четверки, где:

N – алфавит нетерминальных символов;

T – алфавит терминальных символов;

Z – начальный символ грамматики;.

P – есть множество правил вывода – вида $\alpha \Rightarrow \beta$, $\alpha, \beta \in (N+T)^*$, $\alpha \neq \{\varepsilon\}$

Пример

Грамматика целых десятичных чисел

<число>::=<число1>

<число1>::=<число1><цифра>| <цифра>

<цифра>::=0|1|2|...|9

Вывод числа 129

<число> => <число1> =><число1><цифра>

<число1> 9 =><число1><цифра> 9=>.....=> 129

Пусть G грамматика, тогда строка v прямо производит строку w: v => w,

если мы можем записать следующее:

v = x U y; w = x u y (x и y строки)

Если в грамматике существует правило U ::= u, то знак => определяется как знак однократного применения единственного правила.

Пусть G грамматика и v производит строку w: v =>+ w, если мы имеем последовательность прямых преобразований:

$v = u_0 \Rightarrow u_1 \Rightarrow u_2 \Rightarrow ... \Rightarrow u_n = w$, где n>0

Пока в строке имеется хотя бы один нетерминальный символ, преобразование должно быть продолжено.

Строка x называется сентенциальной формой, если она может быть произведена из начального символа грамматики Z.

Предложение есть сентенциальная форма, содержащая только терминальные символы.

Язык L, определяемый грамматикой G[Z], есть множество предложений, которые записываются так: L (G) = { x ∈ T* | Z =>*x}.

2.3. Иерархия языков по Хомскому

Иерархия Хомского – классификация формальных языков и формальных грамматик, согласно которой они делятся на 4 типа по их условной сложности, – предложена профессором Массачусетского технологического института, лингвистом Ноамом Хомским. Одной из основных его идей было то, что смысл предложения может изменяться при изменении структуры предложения. То есть пара <предложение, структура> однозначно определяет смысл.

Определение грамматики, данное в предыдущем разделе, полностью соответствует идее Хомского.

Было доказано, что в общем случае задача грамматического разбора не может быть эффективно решена (то есть нет возможности оценить время работы алгоритма или нет возможности доказать конечность алгоритма), применяя грамматику Хомского в общем виде.

Для решения этой проблемы Хомский предложил четыре семейства грамматик, которые покрывают большинство практических применений, и в то же время для них найдены классы автоматов вывода строк.

Согласно Хомскому, формальные грамматики делятся на четыре типа. Для отнесения грамматики к тому или иному типу необходимо соответствие *всех* ее правил (продукций) некоторым схемам.

Пусть G (V, T, P, Z), T ∈ V, V=N+T, Z ∈ (V–T).

Грамматика типа 0 (укорачивающая грамматика, неограниченная):

u ::= v

u ∈ V^+

v ∈ V*

К типу 0 по классификации Хомского относятся неограниченные грамматики (также известные как грамматики с фразовой структурой). Это все без исключения формальные

грамматики. Этот класс грамматик представляет теоретический интерес, но практически не применяется как таковой. Распознается машинами Тьюринга.

Грамматика типа 1 (контекстно-зависимая грамматика, неукорачивающая):

xUy ::= xuy

$U \in V - T$

$x,y \in V^*$ (результирующая строка может быть пустой),

$u \in V^+$ (результирующая строка не может быть пустой)

Грамматики подобного типа могут использоваться при анализе текстов на естественных языках, однако при построении компиляторов практически не используются в силу своей сложности. Для контекстно-зависимых грамматик доказано утверждение: по некоторому алгоритму за конечное число шагов можно установить, принадлежит цепочка терминальных символов данному языку или нет. Распознается недетерминированными линейно-связанными автоматами.

Грамматика типа 2 (контекстно-независимая или свободная грамматика):

U ::=u

$U \in V - T$

$u \in V^+$

Контекстно-свободные (КС) грамматики широко применяются для описания синтаксиса языков программирования. Распознаются недетерминированными автоматами со стековой организацией памяти.

Грамматика типа 3 (регулярная грамматика):

U ::=n

U ::=nW

U ::=Wn

$n \in T$, $U,W \in V-T$

Регулярные грамматики применяются в лексическом анализе для описания простейших конструкций языка (лексем): идентификаторов, строк, констант, а также языков ассемблера, командных процессоров и др. Распознаются детерминированными конечными автоматами.

3. ЛЕКСИЧЕСКИЙ АНАЛИЗ И ПРОГРАММИРОВАНИЕ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА (СКАНЕРА)

Основные определения

Лексема – неделимая фундаментальная единица языка программирования (ключевое слово, имя переменной, оператор).

Лексический анализатор (сканер) – часть компилятора, выполняющая просмотр исходного текста, распознавание и классификацию лексем.

3.1. Основные задачи лексического анализатора

При любой форме построения программ-переводчиков (компиляторов, трансляторов или интерпретаторов) лексический анализ предложений исходного текста представляет собой самый первый проход по тексту программы. При этом проходе лексический анализатор (в дальнейшем – сканер) путем декомпозиции исходного текста решает несколько важнейших задач по формированию основы следующих фаз перевода. Перечислим основные функции сканера:

1) чтение (сканирование) символов текста программы;
2) распознавание и выделение так называемых единиц синтаксических классов – лексем;
3) построение таблицы идентификаторов с учетом вложенности блоков и процедур;
4) построение таблицы констант;
5) исключение из текста программы комментариев;
6) размещение в памяти строк и тел процедур, представленных машинным кодом или кодом ассемблера;
7) обработка макрокоманд (генерация макрорасширений);
8) формирование файла диагностики лексических ошибок;
9) нумерация строк программы;
10) формирование различных метрик программы.

Число других таблиц, необходимых в процессе компиляции, определенным образом зависит от компилируемого языка. Обычно дополнительно могут использоваться такие таблицы:

- таблица функций,
- таблица меток,
- таблица типов.

Для построения лексического анализатора необходимы грамматика лексем языка и алгоритм распознавания лексем (представленный, например, в виде конечного автомата).

3.2. Регулярные выражения и конечные автоматы

Рассмотрим следующую регулярную грамматику G [Z]:

Z ::= U0 | V1

U ::= Z1 | 1

V ::= Z0 | 0

Здесь Z, U, V – нетерминальные символы;

0, 1 – два терминальных символа.

Легко увидеть, что порождаемый этой грамматикой язык состоит из последовательностей, образуемых парами 01 или 10, т.е. L (G) = $\{B^n \mid n > 0\}$, где B = {01, 10}. Такой язык, по определению, распознается детерминированным конечным автоматом (**КА**).

Чтобы облегчить распознавание предложений грамматики G, нарисуем диаграмму состояний **КА** (рис. 3.1). В этой диаграмме каждый нетерминал грамматики G представлен узлом или состоянием; начальное состояние **КА** – S (предполагается, что грамматика не содержит нетерминала S).

Каждому правилу Q ::= t в диаграмме соответствует дуга с пометкой t, направленная от начального состояния S к состоянию Q.

Каждому правилу Q ::= Rt в диаграмме соответствует дуга с пометкой t, направленная от состояния R к состоянию Q.

Здесь Q, R $\in$ N, t $\in$ T (где N – алфавит нетерминальных символов, T – терминальных).

Пусть дана некоторая цепочка символов *x*, например, 101001# (знак # обозначает признак конца строки). Нетрудно убедиться, что строка *x* есть предложение грамматики G. Мы используем диаграмму состояний, чтобы распознать или разобрать цепочку *x* следующим образом:

1. Перед сканированием **КА** находится в состоянии S. Находясь в S, начать с самого левого символа в цепочке *x* и повторять шаг 2 до тех пор, пока не будет достигнут символ #.

2. Сканировать следующий символ x, перейти по дуге, помеченной этим символом, к следующему текущему состоянию.

3. Если при каком-то повторении шага 2 такой дуги не оказывается, то цепочка *x* не является предложением и происходит останов.

4. Если мы достигаем конца x, то x – предложение тогда и только тогда, когда последнее текущее состояние есть Z.

На каждом шаге (кроме первого) основой является имя текущего состояния, за которым следует входной символ. Символ, к которому приводится основа, будет именем следующего состояния. Чтобы избавиться от проверки на каждом шаге, есть ли дуга с соответствующей меткой, можно добавить еще одно состояние, называемое F (НЕУДАЧА), и добавить все необходимые дуги от всех состояний к F. Добавляется также дуга, помеченная всеми возможными (0,1) символами и ведущая из F обратно в F.

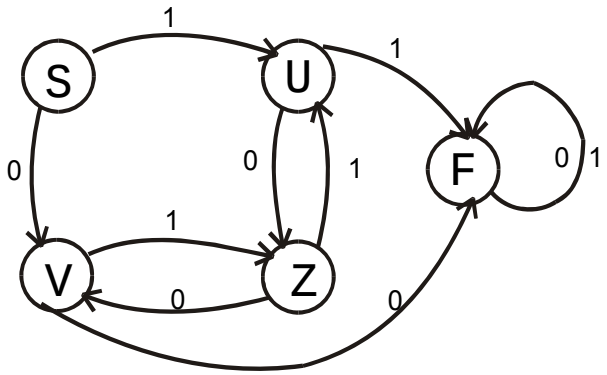


Рис. 3.1. *Диаграмма состояний* ***КА***

Чтобы манипулировать с диаграммами состояний, необходима формализация концепции в терминах состояний, входных символов, начального состояния S, <отображения> M, которое по заданному текущему состоянию Q и входном символе t

указывает следующее текущее состояние, и заключительных состояний, аналогичных состоянию Z в приведенном выше примере.

О п р е д е л е н и е. (Детерминированный) автомат с конечным числом состояний (**КА**) – это пятерка (K, T, M, S, Z), где:

1) K – алфавит элементов, называемых состояниями;
2) T – алфавит, называемый входным алфавитом (символы, которые могут встретиться в цепочке или предложении);
3) M – отображение (или функция) множества K * T во множество K (если M (Q, t) = K, то это означает, что из состояния Q при входной литере t происходит переключение в состояние K);
4) S ∈ K – начальное состояние;
5) Z – непустое множество заключительных состояний, каждое из которых принадлежит K.

3.3. Использование регулярных грамматик и конечных автоматов при программировании сканера

Распознавание лексем – одна из важнейших задач сканера. Для большинства языков программирования можно выделить 5 основных типов лексем:

1) идентификаторы;
2) служебные слова (которые являются подмножеством идентификаторов);
3) целые числа;
4) однопозиционные разделители {+, –, (,), / и т.д.};
5) двухпозиционные разделители {//, /*, **,: = и т.д.}.

Эти лексемы могут быть описаны следующими простыми правилами:

<идентификатор> ::= буква | <идентификатор> буква | <идентификатор> цифра

<целое> ::= цифра | <целое> цифра

<разделитель> ::= + | – | (|) | / |..

<разделитель2> ::= < SLASH >/ | SLASH>* | <AST>* | <COLON >=|..

<SLASH> ::= /

<AST> ::= *

<COLON> ::= :

Определим также, что /*это комментарий*/, который следует «проглотить» при разборе.

Так как правила регулярной грамматики не разрешают использование в правой части правила двух подряд терминалов, то для описания имеющихся в нашем примере двухпозиционных разделителей вводятся дополнительные нетерминальные символы <SLASH>, <AST> и <COLON>.

В этих правилах «буква» представляет любую разрешенную в алфавите языка букву, а «цифра» – любую разрешенную в алфавите цифру. Каждое правило в соответствии с требованиями регулярной грамматики имеет вид U ::= t или U ::= Vt, U, V ∈ N, t ∈ T (где N – алфавит нетерминальных символов, T – терминальных символов).

Начнем реализацию сканера с того, что нарисуем диаграмму состояний некоторого конечного автомата для выделения лексем (рис. 3. 2). Метка D (digit) используется вместо любой из меток 0, 1, 2,..., 9, т.е. D представляет класс цифр. Аналогично метка L (letter) представляет класс букв A, B, ..., Z, а DELIM (delimiter) представляет класс однопозиционных разделителей. Заметим, что символ / (SLASH) не принадлежит к этому классу разделителей, так как он должен обрабатываться особым образом. Некоторые дуги не помечены. Эти дуги будут выбраны, если сканируемый символ не совпадает ни с одним из символов, которыми помечены другие дуги. Например, если мы находимся в состоянии INT (целое), то будем оставаться в этом состоянии до тех пор, пока сканируются цифры. Если же сканируется не цифра, то выходим по дуге, ведущей в OUT. Дуги, ведущие в OUT и ERROR, говорят лишь о том, что обнаружен конец лексемы и необходимо покинуть сканер.

Одна из проблем, которая возникает при переходе в OUT, состоит в том, что в этот момент не всегда сканируется символ, следующий за распознанной лексемой.

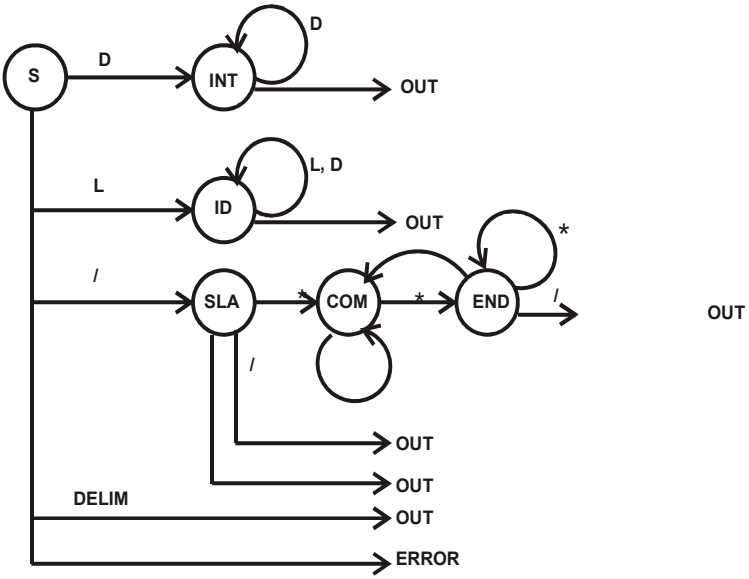


Рис. 3.2. *Диаграмма состояний для сканирования текста программы*

Так, символ выбран при переходе в OUT из INT, но он еще не выбран, если только что распознан разделитель (DELIM). Когда потребуется следующая лексема, необходимо знать, сканировался уже ее первый символ или нет. Это можно сделать, используя, например, переключатель. Будем считать, что перед выходом из сканера следующий символ всегда выбран, то есть находится в переменной CHAR.

3.4. Глобальные переменные и необходимые подпрограммы

Для работы сканера требуются следующие переменные и процедуры:

1) Символьная переменная CHAR. Это глобальная переменная, значением которой всегда будет сканируемый символ исходной программы, т.е. один байт.

2) Целая переменная CLASS. В ней содержится целое число, которое характеризует класс символа, находящегося в CHAR. Будем считать, что CLASS = 1, если сканируемый символ принадлежит D (т.е. цифра). Для букв L CLASS = 2, символ ”/” имеет CLASS = 3, а для класса DELIM CLASS = 4. Определение класса символа (понимание сканером того, что собой представляет двоичная конфигурация байта) решается, как правило, с помощью специальной команды ассемблера, например, TRT в системе IBM/360 и XLAT в IBM PC. Эти команды трактуют двоичный код байта как индекс элемента одномерной таблицы из 256 байт. В каждый соответствующий байт этой таблицы заранее записывается значение класса символа. Так, во всех байтах, чьи индексы соответствуют кодам букв, записывается в нашем случае значение 2. В языках программирования типа PASCAL эту задачу удобнее решить с помощью конструкции CASE.

3) Символьная переменная A, которая будет содержать цепочку (строку) символов, составляющих лексему.

4) GETCHAR – процедура, которая выбирает следующий символ исходной программы и помещает ее в CHAR, а соответствующий класс символа – в CLASS.

5) GETNOBLANK – эта процедура проверяет содержимое CHAR, и если это пробел, то повторно вызывается GETCHAR, пока в CHAR не окажется символ, отличный от пробела.

6) ERROR – процедура обработки ошибок.

3.5. Добавление семантики в диаграмму состояний

Добавим в диаграмму состояний команды, которые обеспечат построение лексем. Как это сделать – показано на рис. 3.3. По существу диаграмма осталась той же, что и на рис. 3.2, но под каждой дугой появились команды, которые надлежит выполнить. К тому же мы явно ввели команду GC, сокращенно обозначив таким образом GETCHAR. Неявно это было сделано и на рис. 3.2.

Мы также вынуждены были несколько расширить блок-схему в той части, где распознается конец идентификатора, чтобы проверить, не является ли набранный идентификатор служебным словом. Процедура LOOKUP осуществляет поиск лексемы, сформированной в переменной A, в таблице служебных и ключевых

слов. Если лексема является служебным или ключевым словом, то соответствующий индекс заносится в глобальную переменную J, в противном случае туда записывается 0. Выражение OUT (C, B) означает возврат к программе, которая вызывала сканер, с двумя величинами C и B в качестве результата.

Под первой дугой, ведущей к состоянию S, записана команда INIT, которая указывает на необходимость выполнения подготовительных действий и начальных установок (инициализации). В данном случае выполняются процедура GETNOBLANK и команда A :=' '. Тем самым в CHAR заносится символ, отличный от пробела, и в ячейку A, в которой будет храниться символ, записывается признак «пусто». Команда ADD означает, что символ из CHAR добавляется к символу в A следующим образом: **A := A CAT CHAR**, где CAT обозначает операцию конкатенации (сцепления). Обратите внимание на тот факт, что при обнаружении комментария или ошибки мы не выходим из сканера.

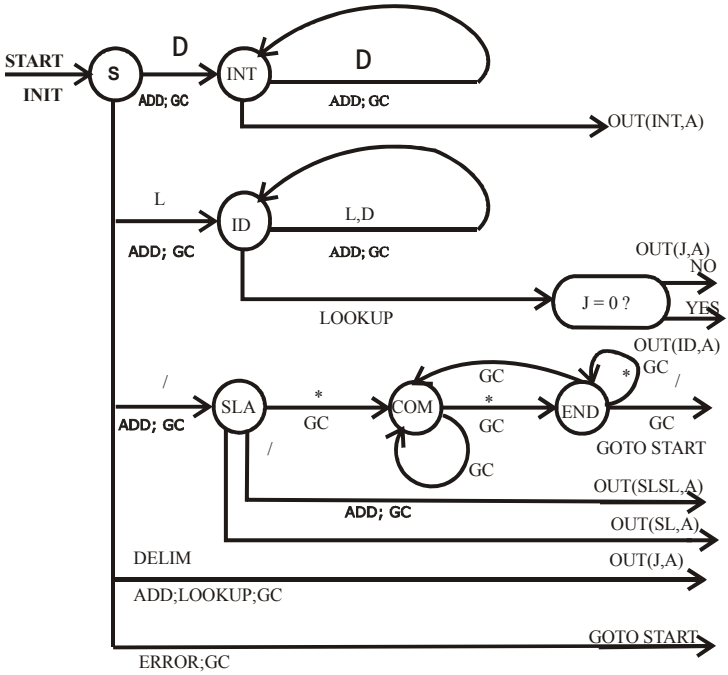


Рис. 3.3. *Диаграмма состояний с семантическими процедурами*

Ниже приводится процедура сканера SCAN, реализующая диаграмму состояний **КА** на рис. 3.3. Процедура в качестве результата распознавания лексем выдает два параметра: в переменной SYN – стандартную кодировку типа лексемы; в SEM – цепочку символов, составляющих лексему. В другом варианте SYN и SEM могли бы быть глобальными ячейками для сканера и для всех тех мест компилятора, в которых вызывается сканер, что обеспечивает более высокую эффективность. В процедуре используется инструкция CASE, которая по номеру класса символа, содержащегося в CHAR, определяет, какую надо выбрать дугу, ведущую из начального состояния S. Язык программирования похож на PASCAL. Так как переменные CLASS, CHAR и A являются глобальными переменными, в процедуре они не описаны.

```
PROCEDURE SCAN (INTEGER SYN,
STRING SEM);

START: GETNOBLANK; A:='';        Строка инициализации.
                                 Инструкция CASE
                                 используется для
CASE CLASS OF                    перехода по дуге из
                                 состояния S в INT, ID,
                                 SLA и т.д.
BEGIN
BEGIN WHILE CLASS=1 DO           CLASS = 1 означает
                                 цифру.
                                 Состояние INT.
BEGIN A:=A CAT CHAR;             Сканировать символы и
                                 добавлять их в A до
GETCHAR;                         тех пор, пока будет
END;                             обнаружена не цифра.
SYN:=$INT;                       Затем сформировать
END;                             параметр SYN.
BEGIN
WHILE CLASS =<2 DO               В CHAR – буква.
BEGIN                            Состояние ID.
A: = A CAT CHAR;                 Сканировать символы до
GETCHAR;                         тех пор, пока будут
END;                             обнаружены не L или не
SYN:=$ID;                        D.

LOOKUP(A);                       Проверить, является ли
IF J<>0 THEN SYN:=J;             идентификатор служебным
END;                             словом (LOOKUP). Если
                                 это так, то
                                 сформировать SYN.
```

```
                                    Состояние SLA.
                                     Если следующий символ
BEGIN A:=CHAR; GETCHAR;              *, то – комментарий.
IF CHAR ='*' THEN                    Сканировать до тех
BEGIN B: GETCHAR;                    пор, пока не
C: IF CHAR <>'*' THEN GOTO           встретится другой
B;                                   символ *.
GETCHAR;                             Следующий символ
IF CHAR <>'/' THEN GOTO C;           может быть / (конец
GETCHAR; GOTO START                  комментария).
IF CHAR ='/' THEN                    Следующий символ / –
BEGIN A:= A CAT CHAR;                комментарий кончился.
GETCHAR; SYN:= $SLSL;                Сканировать следующий
END;                                 символ и начать
ELSE SYN:= $SLASH                    формирование новой
END;                                 лексемы.
BEGIN LOOKUP(A);                     Дуга помечена / –
SYN:= J; GETCHAR;                    выход из состояния
END;                                 SLA.
                                     Это непомеченная дуга
                                     из состояния SLA.
                                     Найти номер
                                     разделителя и занести
                                     его в SYN.
BEGIN ERROR; GETCHAR;                Встретился
GO TO START;                         недопустимый символ.
END                                  Исключить его и
END; SEM:= A;                        продолжить работу.
```

Отметим, что наш сканер всегда строит по возможности наиболее длинный символ. Следовательно, ABC12 – это просто идентификатор, а не идентификатор, за которым следует целое число. Такой принцип вполне оправдан в большинстве случаев, но не во всех. Известным примером является инструкция цикла Фортрана DO10K = 1,20, где DO10K – не идентификатор, а ключевое слово DO, за которым следует номер инструкции 10, за которым, в свою очередь, следует идентификатор K. Очевидно, что для Фортрана наш метод не будет идеальным. Обычно в таких случаях довольно легко написать небольшую программу, которая просмотрит вперед исходную программу и вынесет для таких случаев соответствующее решение.

Теперь настало время усложнить классификацию символов исходной программы. Введем некоторую новую классификацию входных символов исходной программы:

1) DIGIT – символы, которые могут появиться в лексеме <целое> (INTEGER);
2) IDBEG – символы, которые могут появиться в качестве первого символа <идентификатора> (identifier beginner);
3) IDCHAR – символы, которые могут появиться на месте второго, третьего, четвертого, ... символа <идентификатора> (identifier character);
4) IGNORE – символы, которые сканер полностью игнорирует или исключает (такие, как пробел в некоторых реализациях Алгола);
5) INVTER – символы, которые сигнализируют о конце формируемой лексемы, но которые во всех других случаях игнорируются, и сканер должен их исключать (подобно пробелу в некоторых реализациях Алгола).

На диаграмме состояний этот разделитель мы будем обозначать INV для экономии места на схеме.

Упомянутые выше разделители не являются особым предметом рассмотрения, хотя и с ними следует быть внимательным. Нас больше интересует проблема распознавания тех разделителей, которые входят в состав как одно-, так и двухпозиционных разделителей.

Итак, DELIM (разделители) – символы, которые сами являются лексемами, такие, как /, (и). Они, конечно, могут затем использоваться для образования двухпозиционных разделителей или для формирования ключевых слов. Большинство этих классов не должны пересекаться, то есть никакой символ не может принадлежать двум классам. Перекрытие допустимо только для классов DIGIT и IDCHAR. Разобьем разделители DELIM на четыре следующих класса:

1. Разделители, с которых не могут начинаться двухпозиционные разделители или ключевые слова.
2. Разделители, с которых начинается по крайней мере одно ключевое слово, но ни один двухпозиционный разделитель.
3. Разделители, с которых начинается по крайней мере один двухпозиционный разделитель, но ни одно ключевое слово.
4. Разделители, с которых может начинаться и ключевое слово, и двухпозиционный разделитель.

3.6. Диаграмма состояний для новых классов символов

Используя эти 9 новых классов, мы теперь можем изобразить диаграмму состояний (рис. 3.4) для программирования практически универсального сканера. Эта диаграмма имеет одну особенность – возврат, который нам до этого не приходилось использовать. Предположим, что на вход поступают символы .EQ. (в Фортране, например, так обозначается операция сравнения «равно»), причем с "." начинается, таким образом, по крайней мере одно ключевое слово. Кстати, определим возможное различие между служебным и ключевым словами.

Служебным словом назовем такое слово языка, с которого может начинаться предложение или оно является неотъемлемой частью предложения. Типичный пример служебных слов: IF, THEN, ELSE. С ключевого слова предложение начаться не может – типичный пример: ABS, SIN и т.д.

Возникает вопрос: является ли .EQ. одной лексемой или это три лексемы: ".", "EQ" и "." Мы не узнаем этого до тех пор, пока не просмотрим список служебных и ключевых слов. Если .EQ. не является лексемой, процедура GETCHAR «возьмет назад» лексемы с тем, чтобы можно было начать сканирование сначала. С этой целью позиция начального символа отмечается (MARK), и затем, когда возникает необходимость, происходит возврат (BACKUP), и процедура GETCHAR начинает снова с последнего отмеченного символа. Заметим, что возврат назад происходит также и через игнорируемые символы. Поскольку в нашем случае характер возврата строго определен, необходимо помнить лишь о последнем помеченном символе в течение ограниченного промежутка времени. Как только мы выдаем лексему, метка больше не нужна.

Отметим особую роль символа «пробел» в нашем сканере. Он не относится к игнорируемым символам, иначе мы могли бы писать в программе и GOTO, и GO TO (такой подход к трактовке пробела мы уже рассматривали на примере

цикла DO Фортрана). В нашем случае пробел, обнаруженный при нахождении сканера в любом состоянии КА, кроме S, является признаком конца лексемы. Таким образом, то, как разработчик сканера трактует пробел, существенно влияет на правила написания программы! Более того, от этого зависят и правила формирования диагностического файла.

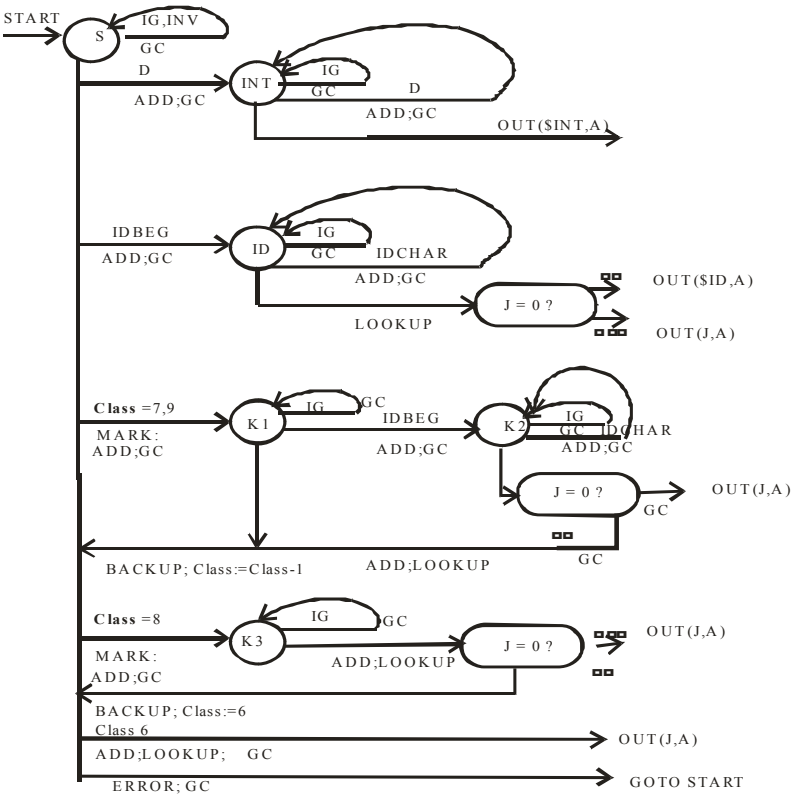


Рис. 3.4. *Диаграмма состояний*

Теперь обратим внимание на следующую проблему. Рассмотрим, например, двухпозиционный разделитель **:=** (присваивание в Паскале). По нашей классификации двоеточие, с которого он начинается, имеет CLASS = 8, и мы попадаем в состояние К3. Находясь в состоянии К3, мы по любому

символу, кроме игнорируемого, переходим к проверке содержания ячейки **A** на предмет нахождения ее в таблице, где мы предварительно расположили все служебные и ключевые слова, а также разделители. Если вторым после двоеточия символом был знак “=”, то мы найдем индекс J разделителя **:=** и таким образом сформируем признак лексемы присваивания. Если после двоеточия мы встретим другой символ, то это будет для нас обозначать, что мы «ошиблись» (конечно же, ошибся программист) при определении класса для ”**:**”, и надо будет трактовать двоеточие как разделитель класса 6. Но процедура LOOKUP не сможет найти записи в таблице, содержащей “**:**”, так как должно выполняться требование о непересечении двух подмножеств разделителей.

А если все-таки в языке имеются разделители, которые могут выступать одновременно в роли однопозиционных и начинать двухпозиционные – например, в Паскале это точка, которая отделяет целую часть числа от дробной, а также присутствует в описаниях, например, массива A [1..10]. Это же можно сказать о косой черте / и других символах. Использование процедуры LOOKUP для формирования признака лексемы в этих случаях принципиально возможно, но не рекомендуется. Проще решить эту проблему непосредственно с помощью соответствующих дуг и дополнительных состояний **КА**, как это сделано на схеме рис. 3.3.

3.7. Примеры построения таблиц лексического анализатора для программного фрагмента на языке Mini PL

```
WCM: PROCEDURE (RATE,START,FINISH);
DECLARE (COST,RATE,START,FINISH) FIXED BINARY (31)
STATIC;
COST= RATE* (START – FINISH) + 2* RATE * (START –
FINISH – 100)
RETURN(COST);
END;
```

Таблица терминальных символов (служебных слов и разделителей)

	Символ	*Разделитель*	*Другие*
1	:	*Да*	
2	;	*Да*	
3	(	*Да*	
4	)	*Да*	
5	,	*Да*	
6	.	*Да*	
7	PROCEDURE	*Нет*	
8	DECLARE	*Нет*	
9	RETURN	*Нет*	
10	END	*Нет*	
	+		
	–	*и т.д.*	

Таблица стандартных символов

Тип	*Индекс*	*Лексическая единица*
IDN	1	WCM
TRM	1	:
TRM	7	PROCEDURE
TRM	3	(
IDN	2	RATE
TRM	5	,
IDN	3	START
TRM	5	,
IDN	4	FINISH
TRM	4	)
TRM	*2*	;
TRM	8	DECLARE
TRM	3	(
IDN	5	COST
TRM	5	,
IDN	2	RATE
и т.д.	*и т.д.*	*и т.д.*

Почему в колонке индекса имеются одинаковые индексы? Просто это индексы для разных таблиц. Программист волен сам решать вопрос – сколько и какие таблицы он будет создавать (объединять) и т.д.

Таблица идентификаторов

	Имя	*Атрибуты*
1	WCM	Заполняется более поздними фазами
2	RATE	
3	START	
4	FINISH	
5	COST	

Таблица литералов (констант)

Литерал	*Основание*	*Формат*	*Точность*	*Другие*	*Адрес*
31	DECIMAL	FIXED	*2*		
2	DECIMAL	FIXED	1		
100	DECIMAL	FIXED	3		

3.8. Некоторые приемы написания сканера на языке C/C++

Вот несколько примеров написания отдельных фрагментов лексического анализа – без прямого использования диаграммы состояния конечного автомата.

Для определения идентификатора можно применить такой вариант:

```
#include <stdio.h>
#include <ctype.h>
main()
{ char in;
  in = getchar();
  if (isalpha(in))  in = getchar();
  else  error();
  while  (isalpha(in)  ||  isdigit(in))  in  =
  getchar();
}
```

Здесь in – значение только что считанного с помощью встроенной функции getchar() символа; встроенные функции isalpha() и isdigit() осуществляют проверку аргумента на предмет принадлежности к буквам и цифрам соответственно; error() выполняет некоторые операции при возникновении ошибки.

Подобным образом можно написать программу для распознавания действительных чисел.

```
#include <stdio.h>
#include <ctype.h>
main ()
{
  char in;
  in = getchar();
  if (in=='+' || in=='-') in = getchar();
  while (isdigit(in))    in = getchar();
  if (in=='.')    in = getchar();
  else     error();
  if (isdigit(in))    in = getchar();
  else    error();
  while (isdigit(in)) in = getchar();
  printf("ok\n");
}
```

Обратите внимание, что возможны три ситуации и три способа их представления в данной программе.

1. Необязательные символы для указания знака числа ("+", "-"). Если их нет – это не ошибка, просто переходим к считыванию следующего символа.

2. Обязательные знаки (десятичная точка и одна цифра после нее). Если их нет – вызывается функция error.

3. Знаки, которые могут появиться нуль или большее число раз (цифра перед точкой или после первой цифры за точкой) – инициируется цикл while для проверки каждого знака без обращения к функции error.

3.9. Контрольные вопросы по теме

1. Назовите основные задачи лексического анализа.
2. Что такое лексема?
3. Какой тип грамматики применяется для описания лексем и в чем его особенности?

4. Как можно применить правила регулярной грамматики к двухпозиционным разделителям?
5. На какие основные классы можно разбить символы исходной программы?
6. Каким образом сканер распознает идентификаторы?
7. Какую информацию и в каком виде выдает сканер после распознавания лексемы?
8. Каким образом решается проблема распознавания неоднозначности разделителей?
9. Что такое таблица стандартных символов?
10. Можно ли двухпозиционный символ поместить в таблицу служебных слов?

4. СИНТАКСИЧЕСКИЙ АНАЛИЗ И ПРОГРАММИРОВАНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА (ПАРСЕРА)

4.1. Основные определения и предпосылки

Синтаксис (грамматика) – четкие и однозначные грамматические правила, которые должны соблюдаться при написании программы.

Синтаксический анализатор (парсер) – программа для проверки правильности грамматического построения конструкций предложений исходного текста программы.

Задача синтаксического анализатора – распознать предложения исходной программы как языковые конструкции, полностью соответствующие используемой грамматике.

Все конструкции современных языков программирования имеют четкие и однозначные грамматические правила (синтаксис), которые должны соблюдаться при написании программы. Для выявления возможных ошибок и контроля правильности построения предложений языка в исходном тексте необходимо иметь:

1) предложение языка, представленное в виде таблицы стандартных символов;
2) грамматику языка, расположенную в памяти (в некоторых методах грамматического разбора это требование может отсутствовать);
3) алгоритм грамматического разбора, реализованный в виде синтаксического анализатора (английский термин – PARSER, которым мы будем пользоваться для краткости).

Рассмотрим эти требования более подробно.

4.2. Таблица стандартных символов

Пусть исходное предложение (правильное!) в программе имеет вид:

IF A > 2.5 THEN B := 0;

С помощью лексического анализатора были распознаны следующие лексемы, содержащиеся в этом предложении:

– служебное слово IF (пусть его индекс в таблице служебных слов равен, например, 15);
– идентификатор (код – 01);
– однопозиционный разделитель ”>“ (индекс в таблице служебных слов равен, например, 30);
– константа (код – 02);
– служебное слово THEN (индекс в таблице служебных слов равен, например, 20);
– идентификатор (код – 01);
– двухпозиционный разделитель “:=“ (код – 05);
– константа (код – 02);
– однопозиционный разделитель”;“ (индекс в таблице служебных слов равен, например, 25).

В результате входной строкой парсера является исходное предложение в виде строки так называемых «стандартных символов»:

[15][01][30][02][20][01][05][02][25]

Очевидно, что эта кодировка и значения индексов в примере абсолютно условны, кроме того, на самом деле эта строка (точнее, фрагмент таблицы стандартных символов) имеет более сложную форму представления. Квадратные скобки здесь применены просто для удобства.

4.3. Грамматика языка

Для описания правил построения предложений большинства языков программирования применяется контекстно-независимая грамматика в ее различных модификациях, записанная в форме Бэкуса–Наура. В нотации Бэкуса–Наура метасимволы (символы, описывающие структуру предложения) заключаются в так называемые метаскобки (угловые скобки), например: <предложение>. Терминальные символы записываются непосредственно в том виде, как мы их видим в тексте программы. Приведем пример простой грамматики для некоторых предложений русского языка:

<предложение> ::= <подлежащее> <сказуемое>
<подлежащее> ::= <существительное> | <местоимение>
<существительное> ::= **дом** | **слон** | **...** | **компьютер**
<местоимение> ::= **я** | **он**
<сказуемое> ::= **идет** | **стоит** | **бежит**

В данном примере комбинация символов "::=" обозначает понятие «заменяется на... », вертикальная черта в соответствии с нотацией Бэкуса–Наура означает альтернативу (возможный выбор, вариант подстановки), а жирным шрифтом записаны слова (лексемы) объектного языка. В соответствии с правилами этой грамматики следующие предложения будут синтаксически правильными:

дом стоит
он идет
компьютер бежит
я идет.....и т.д.

О семантике (смысловом содержании предложения) речь пока у нас не идет, хотя понятно, что при переводе с одного языка на другой мы должны заменить алфавит, лексику (слова), синтаксис (грамматику) одного языка на алфавит, лексику и синтаксис другого языка, но при этом смысл (семантика) должен остаться без изменения! – это классическая задача любого перевода. Приведем еще одну очень важную грамматику – для арифметических выражений:

```
E ::= E+T | E-T | T
T ::= T/F | T*F | F
F ::= (E) | i
```

В качестве метасимволов здесь используются одиночные заглавные буквы, происходящие от английских терминов Expression (выражение), Term (член) и Fa41ctor (коэффициент). Терминальными символами здесь являются знаки операций +, –, *, /, круглые скобки и символ i (обозначает любой идентификатор). Отметим особо, что символ E является в этой грамматике начальным символом.

4.4. Понятие атрибутной грамматики

Большинство программ синтаксического анализа решают две основные задачи:

– проверку соответствия входа парсера контекстно-свободной грамматике, генерирующей расширенное множество языков. Эта часть кода парсера может создаваться «автоматически», исходя из грамматики, специальными программами автоматизации построения компиляторов;

– проверку некоторых необходимых дополнительных ограничений на входные конструкции, например такие, как совместимость типов.

Чтобы контекстно-свободная грамматика предоставляла оба названных аспекта, ее можно улучшить путем введения дополнительных правил. Один из способов, позволяющих это сделать, состоит в использовании атрибутной грамматики (attribute grammar). Приведем пример.

Пусть нам необходимо отслеживать значения выражений, которые задаются грамматикой со следующими правилами:

1. <expr> ::= <expr> + <term>
2. <expr> ::= <term>
3. <term> ::=<term> * <factor>
4. <term> ::= <factor>
5. <factor> ::= constant
6. <factor> ::= (<expr>)

С некоторыми терминалами и нетерминалами грамматики могут связываться атрибуты. Поскольку все атрибуты соответствуют значениям ("value") подвыражений или полных выражений, то естественно обозначить их VAL (там, где значения нельзя спутать). Чтобы не спутать значения различных символов в продукции, будем (где это необходимо) использовать обозначения VAL1, VAL2 и т.д. Атрибуты называются синтезированными, поскольку они используются для передачи значений вверх по синтаксическому дереву или (с другой стороны) для вычисления атрибутных значений, соответствующих левой части продукции, из значений, соответствующих правой части. Атрибуты, которые переносят

информацию в противоположном направлении, то есть вниз по дереву или от левой к правой части продукции, называются наследуемыми.

Ниже приводится пример контекстно-свободной грамматики, которая дополнена атрибутными правилами для определения значений выражений через значения их компонентов. Вертикальная стрелка ↑ перед именем атрибута обозначает, что он является синтезированным, а не наследуемым.

<expr> ↑ VAL ::= <expr> ↑ VAL1 + <term> ↑ VAL2
[правило: ↑ VAL = ↑ VAL1 + ↑ VAL2]
<expr> ↑ VAL ::= <term> ↑ VAL
<expr> ↑ VAL ::=<term> ↑ VAL1 * <factor> ↑ VAL2
[правило: ↑ VAL = ↑ VAL1 * ↑ VAL2]
<term> ↑ VAL ::= <factor> ↑ VAL
<factor> ↑ VAL ::= constant ↑ VAL
<factor> ↑ VAL ::= (<expr> ↑ VAL)

Считается, что если атрибут с одним и тем же именем используется в различных местах продукции, оба экземпляра имеют одинаковое значение.

Атрибутная грамматика для задания правил использования типов является достаточно сложной, так как требует задания эквивалентности наборов информации из таблиц символов, используемых на синтаксическом дереве в качестве атрибутов. Поскольку атрибутная грамматика задает правила соответствия, то ее несложно преобразовать в действия по проверке выполнения правил соответствия. В принципе несложно создать атрибутную грамматику на основе контекстно-свободной программы синтаксического анализа, дополненной определенными действиями.

Атрибутные грамматики также можно использовать для определения метрик исходного кода таким способом, что будет легко создать инструментальные средства для измерения значений метрик. Выразительная сила атрибутных грамматик равна силе грамматик 0-го типа, но при этом первые интуитивно значительно понятнее. Впервые атрибутные

грамматики были определены Д. Кнутом и использовались в качестве основы создания сред программирования, для определения аномалий в программах и как основа парадигмы разработки программного обеспечения. Мы рассмотрим некоторые аспекты реализации атрибутной грамматики в разделе семантического анализа.

4.5. Грамматический (синтаксический) разбор предложения

Основные методы синтаксического разбора представлены четырьмя основными видами:

- нисходящие методы (сверху–вниз, или Top-to-down);
- восходящие методы (снизу–вверх, или Down-to-top);
- смешанные методы (сочетают в себе черты восходящих и нисходящих методов);
- другие (например: алгоритм Early).

4.5.1. Грамматический разбор сверху–вниз

Продемонстрируем применение этой грамматики на примере разбора выражения i – i * (i + i). Разбором мы назовем попытку получить, исходя из начального символа грамматики, данное выражение, применяя правила подстановки (замены), определенные грамматикой. Операцию замены некоторого метасимвола на соответствующую ему правую часть правила обозначим $\Rightarrow$. Итак, начинаем с символа E, и при каждой замене применяем только одно правило:

$E \Rightarrow E - T \Rightarrow E - T*F \Rightarrow T - T*F \Rightarrow T - T*(E) \Rightarrow T - T*(E + T)$
$\Rightarrow T - T*(T + T) \Rightarrow i - T*(T + T) \Rightarrow i - i*(T + T) \Rightarrow i - i*(i + T)$
$\Rightarrow i - i*(i + i)$.

В сентенциальной форме остались одни терминальные символы – разбор закончен.

Данный пример демонстрирует один из двух основных методов грамматического разбора, который называется «Разбор

сверху–вниз» (TOP–DOWN). Основной идеей этого метода является попытка «сгенерировать» разбираемое предложение, начиная с начального символа грамматики Z.

Самый простой способ проделать это – начиная с начального символа грамматики Z, перебирать все правила и попытаться получить в результате исходное предложение. Если попытка удалась, значит предложение построено правильно, если нет, то в предложении есть по крайней мере одна синтаксическая ошибка.

Описанный алгоритм – простейший алгоритм разбора сверху–вниз, и его самым большим недостатком является большая вероятность того, что при выборе какой-либо правой части правила (при наличии альтернатив) программа разбора произведет «тупиковую» замену, после чего ей придется «вернуться назад» и выбрать другую альтернативу из правой части правила.

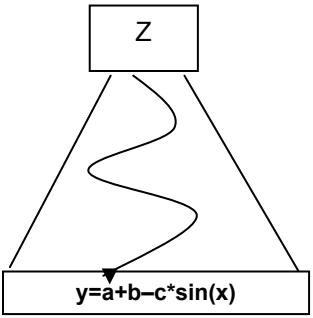


Рис. 4.1. *Разбор сверху-вниз*

В результате большого количества возвратов усложняется сам алгоритм, да и время разбора существенно увеличивается. Поэтому существует «неписаное» правило: программа разбора ни в коем случае не должна прибегать к возвратам! Это необходимо еще и потому, что во время грамматического разбора необходимо связать семантику с синтаксисом, и по мере того как мы будем прогнозировать и находить цели эти символы будут обрабатываться семантически. Вот некоторые примеры «обработки»:

1) при обработке описаний переменных их характеристики помещаются в дескрипторную часть таблицы идентификаторов;

2) при обработке арифметических выражений проверяют, совместимы ли типы операндов.

Если возврат произошел из-за того, что прогнозируемая цель неверна, придется уничтожить результаты семантической обработки, проделанной во время поисков этой цели. Сделать это не так-то просто, поэтому постараемся провести грамматический разбор без возвратов.

Для того чтобы избавиться от возвратов, в компиляторах в качестве контекста обычно используется следующий «незакрытый» символ исходного предложения. Тогда на грамматику налагается следующее требование: если есть альтернативы x | y |... |к, то множества символов, которыми могут начинаться выводимые из x, y, ..., к слова, должны быть попарно различны. То есть если $x \Rightarrow Au$ и $y \Rightarrow Bv$, то $A \neq B$. Если это требование выполнено, можно довольно просто определить, какая из альтернатив x, y или v – наша цель. Заметим, что факторизация оказывает здесь большую помощь. Если есть правило U ::= xy | xv, то преобразование этого правила с помощью приема «факторизация» к виду U ::= x (y | v) помогает сделать множества первых символов для разных альтернатив непересекающимися.

Таким образом, можно выделить основные особенности методов разбора сверху–вниз:

- простота программирования;
- низкое быстродействие;
- возможность отката, если существует несколько правил с одинаковой левой частью (разрешается с помощью предпросмотра);
- возможность зацикливания в случае левой рекурсии правил (разрешается с помощью модификации грамматики).

Пример:

Дана грамматика: S::=T+S|T T::=i * T|i	Исходная строка для разбора: a + b *c	Ожидаемый результат S a + b * c i + i * i

a + b *c	S	
a+b *c	T+S	
a+b *c	i * T + S	
+ b * c	* T + S	Символы «+» и «*» не согласуются, следовательно надо вернуться на какое-то количество шагов назад.
a + b * c	i + S	Возвращаемся на ближайшую замену. Выбирается альтернативный вариант правила для T.
+ b * c b * c b * c b * c * c c c	+S S T i*T *T T i	

4.5.1.1. LL(1)-грамматики

Проблема возвратов при неправильном выборе альтернативы заставляет нас искать другие варианты грамматик, подходящих для нисходящего анализа. Рассмотрим некоторые свойства грамматик, поддерживающих методы нисходящего синтаксического анализа с одним символом предпросмотра – то есть так называемым lookahed. Будем считать, что грамматики являются однозначными.

В данном случае для каждого нетерминала, который находится в левой части нескольких правил, необходимо

найти такие непересекающиеся множества символов предпросмотра, чтобы каждое множество содержало символы, соответствующие точно одной возможной правой части.

С такой идеологией мы уже встречались при лексическом анализе – когда мы анализировали еще один дополнительный символ.

Выбор конкретного правила для замены данного нетерминала будет определяться символом предпросмотра и множеством, к которому принадлежит данный символ.

Объединение различных непересекающихся множеств для заданного нетерминала не обязательно должно составлять алфавит, на котором определен язык. Если символ предпросмотра не принадлежит ни одному из непересекающихся множеств, можно сделать вывод о наличии синтаксической ошибки.

Множество символов предпросмотра, соотнесенных с применением определенного правила, называется ее множеством первых порождаемых символов (director symbol set). Определим вначале следующие две важные категории символов.

1. *Стартовый символ* для данного нетерминала определяется как любой символ (например, терминал), который может появиться в начале строки, генерируемой нетерминалом.

2. *Символ-последователь* для данного нетерминала определяется как любой символ (терминал или нетерминал), который может следовать за нетерминалом в любой сентенциальной форме.

Вычисление множества стартовых символов может быть достаточно трудоемким и вычислительно сложным процессом, и оно всегда выполняется в процессе генерации программы синтаксического анализа, а не при каждом запуске этой программы. Пусть, например, для нетерминала T грамматика содержит только два правила.

T =:: aG| bG

В этом случае имеем следующие множества стартовых символов:

Правило	*Множество стартовых символов*
T =:: aG	{a}
T =:: bG	{b}

В общем случае, если правило начинается с терминала, множество стартовых символов просто состоит из этого терминала. В то же время, если правило не начинается с терминала, то для него все равно нужно вычислить множество стартовых символов. Пусть в рассматриваемой грамматике имеются следующие правила для нетерминала R:

R =:: BG| CH

В этом случае множество стартовых символов нельзя определить «с ходу». В то же время пусть имеются только следующие правила для нетерминала *B*:

B =:: cD| TV

Тогда можно заключить, что множеством стартовых символов для правила R =:: BG будет набор {a, b, c}, состоящий из всех стартовых символов для B.

Введение символа *c* в множество стартовых является очевидным (см. первое правило для B), а введение набора {a, b} объясняется тем, что эти символы являются стартовыми для T. В общем случае ситуация может быть значительно сложнее. Таким источником сложностей можно назвать нетерминалы, которые могут генерировать пустые строки.

Множество первых порождаемых символов правила выбирается как множество всех терминалов, которые, выступая как символы предпросмотра, указывают на использование данного правила. Таким образом, множество первых порождаемых символов для правила

A =:: BC

будет включать все символы-последователи A, а также стартовые символы BC.

Ниже приводятся множества первых порождаемых символов для различных правил некоторой грамматики.

Правило	*Множество первых порождаемых символов*
S =:: Ty	S – начальный символ грамматики
T =:: AB	{a, b, y}
T =:: sT	{s}
A =:: aA	{a}
A =:: ε	{b, y}
B =:: bB	{b}
B =:: ε	{y}

Существует алгоритм поиска множеств первых порождаемых символов для всех правил грамматики. Сложность этого алгоритма в основном связана с тем, что нетерминальные символы могут генерировать пустые строки. После вычисления всех множеств первых порождаемых символов их можно проверить на предмет пересечения.

LL(1)-грамматику можно определить как грамматику, в которой для каждого нетерминала, появляющегося в левой части нескольких правил, множества первых порождаемых символов всех правил, в которых появляется этот нетерминал, являются непересекающимися.

Термин “LL(1)” имеет следующее происхождение: первое L означает чтение слева (Left) направо, второе L означает использование левых (Leftmost) порождений, а 1 – один символ предпросмотра.

Описанная выше грамматика, очевидно, является LL(1)-грамматикой, поскольку множества символов предпросмотра для T, A и B не пересекаются. Если вычислены все множества первых порождаемых символов для всех возможных правых частей правил, то языки, которые описываются LL(1)-грамматикой, всегда анализируются детерминированно, то есть без необходимости отменять правило после его применения (возврат).

Существуют более распространенные классы грамматик, которые могут использоваться для детерминированного нисходящего анализа, но обычно используются именно LL(1)-грамматики. Недетерминированный нисходящий анализ, основанный на откате (backtracking), уже не считается эффективной процедурой, хотя в начале эры компиляторов он широко использовался в языках, подобных FORTRAN. Грамматики LL(k), требующие k символов предпросмотра для различения альтернативных правых частей, также уже не считаются практичными с точки зрения синтаксического анализа.

Кроме всего прочего, надо различать LL(1)-языки и LL(1)-грамматики. LL(1)-язык – это язык, который можносгенерировать посредством LL(1)-грамматики. Отсюда следует, что для любого LL(1)-языка возможен нисходящий синтаксический анализ с одним символом предпросмотра.

Кроме того, существует алгоритм определения, относится ли данная грамматика к классу LL(1), поэтому грамматику можно проверить на “LL(1)-ность”, прежде чем создавать на ее основе программу синтаксического анализа. В то же время не существует алгоритма определения, относится ли данный язык к классу LL(1), то есть имеет он LL(1)-грамматику или нет. Это означает, что не-LL(1)-грамматика может иметь или не иметь эквивалентную LL(1), генерирующую тот же язык, и не существует алгоритма, который для данной произвольной грамматики определит, является ли генерируемый ею *язык* LL(1) или нет. В общем случае задача является неразрешимой в том же смысле, как неразрешимы задача определения однозначности языка и проблема остановки для машин Тьюринга.

Приведенный выше результат является важным, поскольку имеются грамматики, не являющиеся LL(1), которые, тем не менее, генерируют LL(1)-языки, то есть грамматики имеют эквивалентные LL(1)-грамматики. Это означает, что грамматики часто нужно преобразовывать, прежде чем использовать с методами нисходящего синтаксического анализа. Фактически грамматики, которые обычно используются в определениях языков или в учебниках, редко являются LL(1) и, следовательно, не могут непосредственно использоваться для эффективного нисходящего анализа. Существует еще одно свойство грамматики, которое (если оно присутствует) препятствует тому, чтобы грамматика была LL(1), и это левая рекурсия.

4.5.1.2. Расположение грамматики в памяти

Для грамматического разбора предложения, как это было продемонстрировано выше, необходимо каким-либо способом расположить грамматику предложений в оперативной памяти – так как алгоритм постоянно обращается к отдельным компонентам правил.

На практике приняты несколько вариантов этого размещения элементов правил в памяти. И самый простой из них – это расположение правил в одномерном массиве.

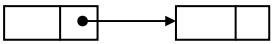
Пример

Грамматика:
Z::=E + T | T
Вариант расположения в виде одномерного массива

Z	E	+	T	|	T	#	E	T	...

Знак решетки # определяет завершение правой части правила, а знак | определяет альтернативу. Первый знак в массиве или после решетки – это нетерминальный символ левой части правила.

В виде списка



.....

В виде структуры

NAME		
DEF	ALT	SUC

Здесь DEF – указатель на первый символ в правой части правила для этого символа. Ноль, если терминал;

ALT – указатель на первый символ в правой части альтернативного правила для этого символа;

SUC – указатель на следующий терминальный/ нетерминальный символ.

Безусловно, расположение грамматики в виде структуры – достаточно сложно для построения, но навигация по правилам существенно облегчена.

Пример

Грамматика:
E ::=E <aop> T|T
T ::=T<mop>F|F
F ::=i| (E)
<aop> ::=+|-
<mop> ::=*|/

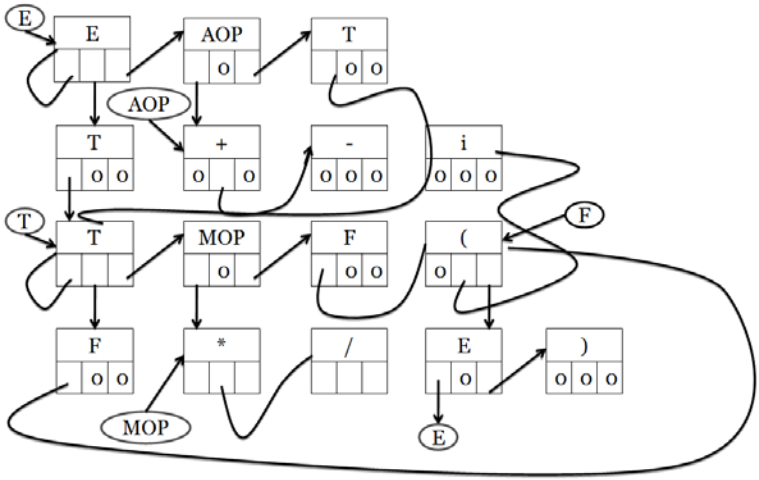


Рис. 4.2. *Список структур, описывающий грамматику*

4.5.1.3. Метод рекурсивного спуска (recursive descent)

Метод рекурсивного спуска может применяться только к LL(1)-грамматикам, поэтому вполне возможно, что исходную грамматику придется несколько преобразовать. Преобразования грамматики не всегда являются очевидными. Простейшим путем их определения является рассмотрение *языка*, генерируемого правилами, подлежащими преобразованию. В то же время данный тип преобразования является настолько общим, что его легко определить и выполнить вручную или автоматически. Данный процесс часто называется *факторизацией*, по аналогии с соответствующим алгебраическим процессом.

В самой общей форме это может быть описано следующим образом. В некоторых компиляторах синтаксический анализатор содержит по одной рекурсивной процедуре для каждого нетерминала U. Каждая такая процедура осуществляет разбор фраз, выводимых из U. Процедуре сообщается, с какого места данной программы следует начинать поиск фразы, выводимой из U. Следовательно, такая процедура целенаправленная, или прогнозирующая.

Мы предполагаем, что сможем найти такую фразу. Процедура ищет эту фразу, сравнивая предложение в указанном

месте с правыми частями правил для U и вызывая по мере необходимости другие процедуры для распознавания промежуточных целей.

Чтобы проиллюстрировать этот способ, запишем процедуры для нетерминальных символов такой грамматики:

```
    <state>::=var>:=<exp>|IF<exp>THEN<state>[ELSE<
state>]
    <var>::=i [(<exp>)]
    <exp>::=<term>{+ <term>))
    <term>::=<fact>{* <fact>}
    <fact>::=<var> | (<exp>)
```

Здесь рассматривается 2 типа предложений – IF (полная и укороченная версия) и присваивание. Причем разрешены простые и индексированные переменные (с одним индексом). Отметим, что <exp>, стоящий после IF, требует некоторого разъяснения и здесь является примером демонстрационного вида.

Мы полагаем, что сможем провести разбор без возвратов. Для того чтобы возвратов не было, в качестве контекста используется единственный символ, следующий за уже разобранной частью фразы.

Будем использовать следующие переменные и процедуры:

1. Глобальная переменная NS всегда содержит тот символ исходной программы, который будет обрабатываться следующим. При вызове процедуры для поиска новой цели первый символ, который она должна исследовать, уже находится в NS.

2. Подобно этому, перед тем как выйти из процедуры с сообщением об успехе, символ, следующий за уже обработанной подцепочкой, помещается в NS.

3. Процедура SCAN (лексический анализатор) готовит код очередной лексемы исходной программы и помещает его в NS.

4. Программа ERROR вызывается в тех случаях, когда обнаружена ошибка. Она печатает сообщение и передает управление в точку вызова. После возврата мы продолжим работу так, как будто бы никакой ошибки не было. Однако известно, что некоторые современные коммерческие компиляторы обычно при обнаружении хотя бы одной синтаксической ошибки прекращают работу. Такое поведение вполне

оправданно, так как очень часто одна синтаксическая ошибка является первопричиной лавины последующих ошибок, которые являются просто следствием самой первой – исправление первой ошибки влечет за собой исчезновение многих последующих. Все это затрудняет отладку. С другой стороны, останов компиляции после единственной обнаруженной ошибки существенно увеличивает время отладки.

Итак, для того чтобы начать синтаксический анализ инструкции, мы обращаемся к программе SCAN, которая поместит первый символ в NS, а затем вызываем процедуру STATE.

Ниже приводятся тексты всех процедур разбора для каждого нетерминального символа грамматики. Процедуру ERROR программист создает самостоятельно в соответствии с принципами, которых он придерживается при создании диагностических сообщений.

```
    PROCEDURE STATE;        Подпрограмма для <state>.
                            Мы обычно полагаем, что можно
    IF NS = 'IF' THEN       определить вид инструкции по
    BEGIN SCAN; EXPR;       первому символу.
    IF NS <> 'THEN'         Взять первый символ
    THEN ERROR ELSE         выражения и вызвать процедуру
                            для обработки выражения.
                            Следующим символом должен
                            быть <THEN>, затем –
    BEGIN SCAN; STATE;      инструкция. Рекурсивный вызов.
    IF NS='ELSE' THEN       Если встретится символ <ELSE>,
    BEGIN    SCAN;    STATEпроизвести его анализ.
END
    END                     Это не условная инструкция.
    END                     Должна быть инструкция
    ELSE                    присваивания. Перейти к разбору
    BEGIN VAR;              переменной, проверить, есть ли
    IF NS <>':=' THEN ERRORсимвол :=, и затем анализировать
    ELSE     BEGIN     SCAN;выражение.
EXPR                        Конец процедуры <state>.
END
```

```
    END;                          Подпрограмма для <var>.
                                  В NS должен содержаться i.
    PROCEDURE VAR;                Взять символ, следующий
    IF NS <> 'i' THEN ERROR за i. В том и только в том случае,
    ELSE BEGIN SCAN;              когда это открывающая скобка,
    IF NS ='(' THEN BEGINперейти к разбору выражения и
SCAN;                             проверить, следует ли за выра-
    EXPR;                         жением закрывающая скобка.
    IF NS <>')' THEN ERROR Занести в NS символ, следующий
    ELSE SCAN                     за обработанной конструкцией.
    END
    END;                          Подпрограмма для <EXPR >
                                  Выражение должно начинаться
    PROCEDURE EXPR;               термом. После этого ожидается
    BEGIN TERM;                   любое количество конструкций
    WHILE NS = '+' DO             вида "+<term>".
    BEGIN SCAN; TERM ENDВозврат, если следующий символ
                                  не "+".
                                  Подпрограмма для <term>.
                                  Процедура аналогична
                                  процедуре для <exp> и не
    PROCEDURE TERM;               требует пояснений.
    BEGIN FACTOR; WHILE
    NS ='*' DO
    BEGIN SCAN; FACTOR            Подпрограмма для <fact>.
    END                           По первому символу выбирается
    END;                          альтернатива. Это выражение,
                                  заключенное в скобки.
    PROCEDURE FACTOR;             Проверить наличие
    IF NS = '(' THEN              закрывающей скобки и
    BEGIN SCAN; EXPR;             сканировать следующий символ.
    IF NS <> ')' THEN ERROR Это переменная.
    ELSE SCAN
    END
    ELSE VAR;
```

Одним из основных ограничений анализа методом рекурсивного спуска, как и других методов LL(1)-анализа, является необходимость преобразования грамматики. При этом

применяются два типа преобразования – удаление левой рекурсии и факторизация. Хороший обзор этих преобразований может быть найден у Хантера [6].

Метод рекурсивного спуска применим в том случае, если каждое правило грамматики для символа $U \in N$ имеет вид:

- либо $U\alpha \rightarrow$, где $\alpha \in (T+N)^*$, и это единственное правило вывода для этого нетерминала;
- либо $U \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid ... \mid a_n\alpha_n$,

где $a_i \in T$ для всех $i = 1,2,...,n$; $a_i \neq a_j$ для $i \neq j$; и $\alpha_i \in (T+N)^*$, т. е. если для нетерминала U правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть различными.

Если правила вывода имеют такой вид, то рекурсивный спуск может быть реализован по вышеизложенной схеме.

При описании синтаксиса языков программирования часто встречаются правила, описывающие последовательность однотипных конструкций, отделенных друг от друга каким-либо знаком-разделителем (например, список идентификаторов при описании переменных, список параметров при вызове процедур и функций и т.п.).

Общий вид этих правил:

L ::= a | a,L (либо в сокращенной форме L ::= a {,a})

Формально здесь не выполняются условия применимости метода рекурсивного спуска, так как две альтернативы начинаются одинаковыми терминальными символами.

Действительно, в цепочке a,a,a,a,a из нетерминала L может выводиться и подцепочка a , и подцепочка a,a , и вся цепочка a,a,a,a,a. Неясно, какую из них выбрать в качестве подцепочки, выводимой из L. Если принять решение, что в таких случаях будем выбирать самую длинную подцепочку (что и требуется при разборе реальных языков), то разбор становится детерминированным.

4.5.2. Грамматический разбор снизу–вверх

Грамматический разбор снизу–вверх начинается с конечных узлов дерева грамматического разбора и в процессе выполнения пытается объединить их построением узлов более высокого уровня до тех пор, пока не будет достигнут

корень дерева Z. В этом смысле разбор снизу–вверх можно противопоставить разбору сверху–вниз.

На каждом шаге свертки подцепочка, которую можно сопоставить правой части некоторого правила вывода, заменяется символом левой части этого правила вывода (свертка).

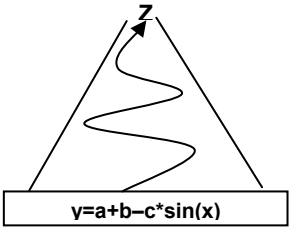


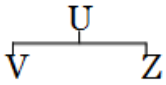
Рис. 4.3. *Разбор снизу–вверх*

Для осуществления разбора необходимо выделить комбинацию символов из правой части правила и свернуть в левую часть (предлог).

Пример

U::=VZ , т.е. VZ сворачивается в U (U=>VZ).

С помощью дерева это можно показать так:



То есть VZ сворачивается в U.

Таким образом, у нас встает проблема: с чего начать разбор, за что, так сказать, зацепиться? Для этого и существует объект, называемый нами «предлог» «основа». В английской терминологии это слово “Handle” – ручка.

4.5.2.1. Применение отношений в грамматиках

Символы "=>" и ":=>+", определенные в разд. 2.3, являются примерами отношений между цепочками. Вообще говоря, (бинарным) отношением на множестве является любое свойство, которым обладают (или не обладают) любые два упорядоченных символа этого множества. Другим примером является отношение МЕНЬШЕ ЧЕМ (<), опреде-

ленное на множестве целых чисел: i<j тогда и только тогда, когда j-i является положительным ненулевым целым числом. Нематематическим является отношение РЕБЕНОК на множестве людей. Некто A либо является ребенком B, либо нет.

Мы используем инфиксные обозначения для отношений: если между элементами c и d некоторого множества имеет место отношение (**R**elation), то мы пишем cRd. (Заметим также, что важен порядок следования в строке двух объектов: из cRd автоматически не следует dRc.)

Можно также рассматривать отношение как множество упорядоченных пар, для которых данное отношение справедливо: (c, d) ∈ R тогда и только тогда, когда cRd.

Мы говорим, что отношение P *включает* другое отношение R, если из (c, d) ∈ R следует (c, d) ∈ P.

Отношение, *обратное* отношению R, записывается как R^{-1} и определяется следующим образом:

c R^{-1} d тогда и только тогда, когда dRc.

Отношение, обратное отношению БОЛЬШЕ ЧЕМ, есть МЕНЬШЕ ЧЕМ. Отношение, обратное отношению РЕБЕНОК, есть РОДИТЕЛЬ.

Отношение R называется *рефлексивным,* если cRc справедливо для всех элементов c, принадлежащих множеству. Например, отношение МЕНЬШЕ ЧЕМ ИЛИ РАВНО (≤) является рефлексивным (i ≤ i для всех действительных чисел i), тогда как отношение МЕНЬШЕ ЧЕМ таковым не является.

Отношение R называется *транзитивным,* если aRb и bRc влечет за собой a R c. Отношение МЕНЬШЕ ЧЕМ над целыми числами является транзитивным. Отношение РЕБЕНОК не является транзитивным; если Джон сын Джека и Джек сын Джила, то, очевидно, Джон не является сыном Джила.

Для любого отношения R определим новое отношение R^+ и назовем его *транзитивным замыканием* R. Оно называется так потому, что включается в любое транзитивное отношение, которое включает в себя R. Другими словами, предположим, что P – транзитивное отношение, которое включает в себя R : из (c, d) ∈ R следует (c, d) ∈ P. Тогда P также включает в себя R^+.

Прежде всего для двух заданных отношений R и P, определенных на одном и том же множестве, введем новое отношение RP, названное *произведением* R и P: cRPd тогда и только тогда, когда существует такое ***e***, что cRe и ePd. Умножение бинарных отношений ассоциативно: R (PQ) = (RP) Q для любых отношений R, P и Q.

В качестве примера рассмотрим отношения R и P над целыми числами, определенные следующим образом:

aRb тогда и только тогда, когда b = a + 1,

aPb тогда и только тогда, когда b = a + 2.

Очевидно, что aRPb тогда и только тогда, когда есть такое ***c***, что aRc и cPb, то есть тогда и только тогда, когда b = a + 3.

Используя произведение, определим теперь степени отношения R следующим образом:

$R^1 = R$, $R^2 = RR$, $R^n = R^{n-l}R = RR^{n-l}$ для $n > 1$. Определим R^0 как *единичное* отношение:

aR^0b тогда и только тогда, когда a = b.

И, наконец, *транзитивное замыкание* R^+ отношения R определяется следующим образом:

aR^+b тогда и только тогда, когда aR^nb для некоторого $n > 0$.

Очевидно, если aRb, то aR^+b. Ясно, почему для транзитивного замыкания выбрано обозначение R^+; когда отношения рассматриваются как множества упорядоченных пар, мы имеем

$R^+ = R^1 \cup R^2 \cup R^3 \ldots$

Определим *рефлексивное транзитивное замыкание* R* отношения R как aR*b тогда и только тогда, когда a = b или aR^+b. Таким образом, $R^* = R^0 \cup R^1 \cup R^2 \ldots$

Чем же вызван такой интерес к отношениям? Во-первых, теперь должно быть очевидно, что символ =>+ обозначает не что иное, как транзитивное замыкание отношения =>. И, во-вторых, с грамматиками связано несколько важных символьных множеств, которые в дальнейшем нам придется так или иначе применять. Эти множества легко определяются

в терминах совсем простых отношений и их транзитивных замыканий.

усть задана грамматика и нетерминальный символ U, нам необходимо знать множество головных символов в цепочках, выводимых из U. Таким образом, если U =>+ Sx, то цепочка Sx выводима из U и S принадлежит множеству. Назовем это множество *голова* (U) и определим его формально следующим образом:

голова(U) :: = {S | U =>+ S . . .}

(Напомним, что тремя точками «. . .» обозначается цепочка, возможно, пустая, которая в данный момент нас не интересует.) Фактически можно представить термин «голова» как множество символов, с которых начинается правая часть правила для U.

В большинстве случаев такое определение могло бы быть удовлетворительным. Заметим, однако, что в определении используется отношение =>+, заданное на бесконечном множестве цепочек в словаре V, и хотя само множество голова (U) конечно, при его построении могут возникнуть трудности. Поэтому мы переопределим множество голова (U), используя другое отношение, заданное на конечном множестве следующим образом. Определим отношение FIRST (первый) для конечного словаря V грамматики следующим образом:

U FIRST S тогда и только тогда, когда существует правило U : : = S . . . .

Затем, согласно определению транзитивного замыкания, имеем

U FIRST* S тогда и только тогда, когда существует непустая последовательность правил U : := S1 . . ., S1 : := S2 . . ., . . ., Sn : := S . . .

Из этого, очевидно, вытекает, что

U FIRST+ 3 тогда и только тогда, когда U =>+ S . . .

Заметим, что если отношение FIRST+ рассматривать как множество, то оно полностью определяет множество голова (U) для всех символов U словаря V: голова (U) = {S | (U, S) в FIRST+}.

Пример. Чтобы проиллюстрировать оба отношения FIRST и FIRST+, выпишем несколько правил и справа от каждого из них укажем отношение FIRST, выведенное из этого правила.

Правило		Отношение
A	::=Af	A FIRST A
A	::=B	A FIRST B
B	::=DdC	B FIRST D
B	::=De	B FIRST D
C	::=e	C FIRST e
D	::=Bf	D FIRST B

Таким образом, мы имеем следующие пары в FIRST+:

(A, A) (A, B) (A, D) (B, B) (B, D) (D, B) (D, D) (C, e)

Следовательно, головными символами цепочек, выводимых из A, будут A, B и D, из B – B и D, из C – e.

Есть еще три других множества, которые мы хотим здесь определить. Они могут быть определены тем же путем, что и FIRST+. Первое из них – это множество символов, которыми оканчиваются цепочки, выводимые из некоторого символа U. Оно определяется для всех символов U через отношение LAST+, являющееся транзитивным замыканием отношения LAST (**последний**):

U LAST S тогда и только тогда, когда существует правило U ::= ...S.

Второе множество – это множество внутренних символов цепочек, выводимых из нетерминала U. Оно определяется через отношение WITHIN+, являющееся транзитивным замыканием отношения WITHIN (**внутри**).

U WITHIN S тогда и только тогда, когда существует правило

U ::= ...S... .

И, наконец, определим множество символов S, которые выводимы из нетерминала U. Оно определяется через отношение SYMB+, являющееся транзитивным замыканием отношения SYMB (симв):

U SYMB S тогда и только тогда, когда существует правило U : : = S .

4.5.2.2. Отношения предшествования

Исходя из описанных выше положений построения отношений FIRST, LAST и WITHIN, мы можем предложить систему отношений между символами, стоящими рядом в некоторой строке.

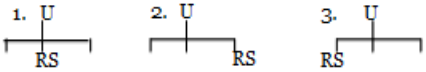
Пусть имеем два произвольных символа R,S $\in$ (N + T). Рассмотрим некоторую строку «....RS...», здесь: символ R есть левый символ, символ S есть рядом стоящий правый символ.

Для того чтобы выделить предлог и произвести свертку, необходимо построить систему отношений между R и S.

1. R·= S – обе лексемы имеют один уровень предшествования и должны рассматриваться как составляющие одной конструкции языка и сворачиваться одновременно. Отношение носит название: равенство по предшествию.

2. R ·> S – R предшествует S (выше по предшествованию). R находится в предлоге, а S – нет. R сворачивается раньше S.

3. R <·S – R – не предшествует S (ниже по предшествованию). S находится в предлоге, а R – нет. S сворачивается раньше R.



Если отношение предшествования не существует, то такие лексемы не могут находиться рядом.

Пример

A + B * C – D

«+» имеет более низкий уровень предшествования, чем * (+ <· *)

«*» имеет более высокий уровень предшествования, чем - (* ·> -)

Тогда, A + B * C – D

<· ·>

Следовательно, подвыражение B*C должно быть обработано ранее.

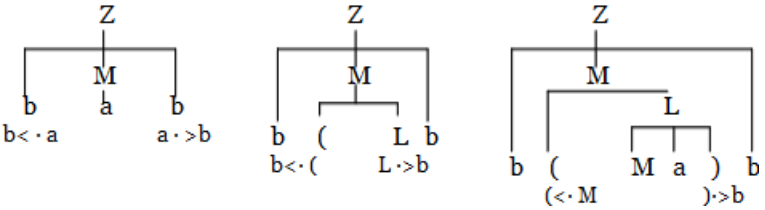
Пример

Дана грамматика:
Z::= bMb
M::= (L|a
L::= Ma)

Запишем элементы грамматики с равными отношениями предшествования:

b·= M; M·= b; (·= L; M·= a; a·=);

Построим некоторые возможные сентенциальные формы:



Соответствующая матрица предшествования выглядит так:

R\S	Z	b	M	L	a	(	)
Z							
b			·=		<·	<·	
M		·=			·=		
L		·>			·>		
a		·>			·>		·=
(			<·	·=	<·	<·	
)		·>			·>		

Рис. 4.4. *Матрица предшествования*

Можно выделить основные особенности метода:

- универсальность;
- не чувствителен к левой рекурсии и неоднозначности грамматики;
- сложность реализации (необходимы специальные алгоритмы для автоматизации построения матрицы предшествия);
- строится на базе стека и таблицы предшествия.

Метод предшествия применим к языковым конструкциям, построенным на базе грамматик предшествования.

4.5.2.3. Определение грамматики предшествования

Грамматика G называется простой грамматикой предшествования типа (1,1), если:

1. Не более чем одно отношение имеется между любыми двумя рядом стоящими символами.
2. Никакие два правила не имеют одинаковых правых частей.

В теории рассматриваются и грамматики с иным предшествием, например грамматики

(n,m) – расширенного предшествования;

слабого предшествования;

смешанной стратегии предшествования;

операторного предшествования.

Алгоритм разбора на основе простого предшествия.

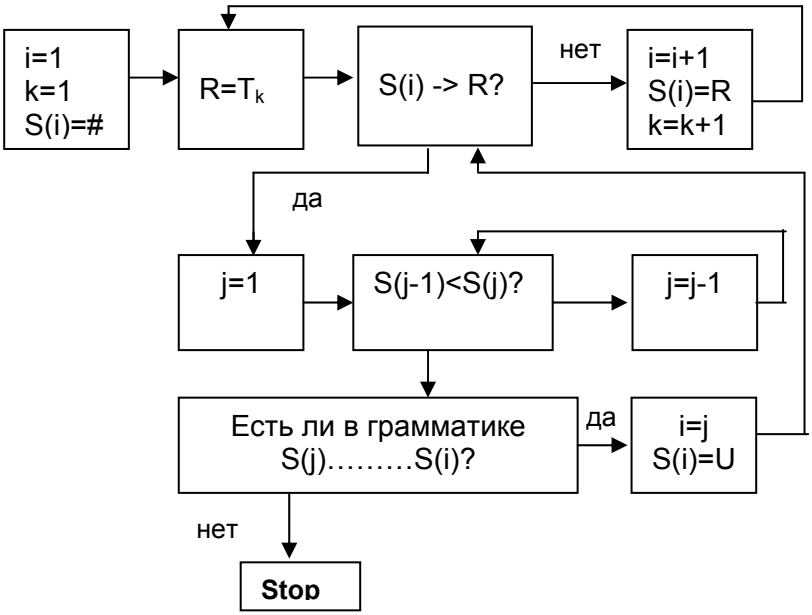


Рис. 4.5. *Алгоритм разбора для грамматики предшествия*

Здесь i – индекс стека S, k – индекс символа в строке разбора, R – текущий символ в строке разбора.

Предложение будет соответствовать грамматике, если в конце разбора i = 2, S(i) = Z и R =# .

Пример

Ниже приводится разбор предложения #b(aa)b#, соответствующего грамматике. Знаки **<-** **÷** **->** обозначают предшествие, равенство по предшествию и непредшествие соответственно.

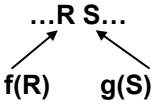
Шаг	Стек	Отношение	R	Строка
1	#	<-	b	(aa)b#
2	#b	<-	(	aa)b#
3	#b(	<-	a	a)b#
4	#b(a	->	a	a)b#
5	#b(M	÷	a	a)b#
6	#b(Ma	÷	)	b#
7	#b(Ma)	->	b	#
8	#b(L	->	b	#
9	#bM	÷	b	#
10	#bMb	->	#	
11	#Z	->	#	

4.5.2.4. Функции предшествования

Необходимость использования матрицы предшествия является одним из основных недостатков рассмотренного выше метода, так как процесс построения подобных матриц достаточно трудоемкий. Кроме того, матрицы предшествий могут быть достаточно большими по размеру и занимать много места в памяти. Поэтому возникла идея минимизации (линеаризации) матрицы предшествия.

Для решения этой проблемы можно воспользоваться функциями предшествия, с помощью которых можно определить отношения предшествования для символов некоторым иным способом, не обращаясь к матрице.

Пусть имеем два рядом стоящих символа R и S. Определим функцию f(R) как функцию для левого символа и g(S) – как функцию для правого символа.

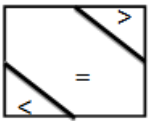


Функции предшествования строятся по предварительно построенной матрице. Функции формируются согласно простым правилам:

– Если R·=S, то f(R)=g(S)
– Если R·>S, то f(R)>g(S)
– Если R<·S, то f(R)<g(S)

Рис. 4.6. *Определение функций предшествия*

Функции можно построить только для матриц, которые приводимы к трехдиагональному виду:



Рассмотрим алгоритм построения функций предшествования:

1. Строится граф, содержащий 2n вершин, где n – количество символов в алфавите (N+T), каждая вершина соответствует одному символу.
2. Рассмотрим пары R(i) и S(j).
   – Если S(i)·=S(j), то проводим дугу из S(i) в S(j) и из S(j) в S(i)
   – Если S(i)->S(j), то проводим дугу из S(i) в S(j)
   – Если S(i)<-S(j), то проводим дугу из S(j) в S(i).
3. Подсчитывается количество вершин, доступных из каждой вершины с помощью переходов + 1. Это и будет значением соответствующей функции для вершины.

Для предыдущей матрицы предшествия значения функций предшествия могут быть представлены так:

	Z	b	M	L	a	(	)
f	1	4	7	8	9	2	8
g	1	7	4	2	7	5	9

4.5.2.5. Операторное предшествование

Каждому оператору языка ставится уровень предшествования. На основе этого уровня выносится решение о том, какой оператор должен выполняться раньше. В терминах синтаксического дерева это означает, что оператор с более высоким уровнем предшествования ближе к листьям дерева и, соответственно, дальше от корня.

В рамках этого метода анализируемое предложение сканируется слева направо, пока не будет найдено подвыражение, операторы которого имеют более высокий уровень предшествования. Это подвыражение распознается в терминах правил вывода используемой грамматики и заменяется соответствующим нетерминалом. Процесс продолжается, пока предложение не будет распознано целиком.

Матрица предшествия операторов меньше полной матрицы предшествия. Рассматриваются отношения только между операторами и обобщенным символом идентификатора - i.

Пример:

В реализации этого метода используются 2 стека: стек операторов OPTOR и стек операндов OPAND. В стеке операторов может находиться:

- признак начала/конца предложения «#»,
- символы стека S(i).

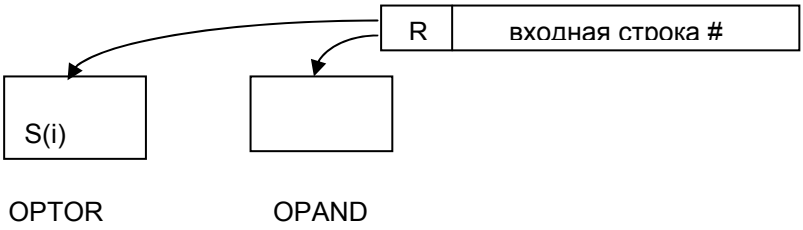


Рис. 4.7. *Алгоритм разбора для операторного предшествия*

Алгоритм метода:

1. Если R – идентификатор, то он помещается в стек операндов.

2. Если f(S(i))<=g(R), то R помещается в стек операторов.

3. Если f(S(i))>g(R), то вызывается некоторая семантическая процедура для S(i), с ее помощью из стека удаляются и, возможно, некоторые другие операторы и/или операнды. Переход к шагу 1.

Пример:

Разберем строку #A+B+C#. Переменная Ti хранит результат действия семантической процедуры.

Шаг	**Строка**	**Стек операторов**	**Стек операндов**
1	A+B+C#	#	
2	+B+C#	#	A
3	B+C#	+ #	 A
4	+C#	+ #	B A
5	+C#	#	T1=A +B
6	C#	+ #	T1
7	#	+ #	C T1
8			T2=C +T1

- Преимущество: удобно реализовывать семантические процедуры.
- Недостатки:
- необходимость помещать одноместные операторы отдельно, то есть (-B+C) содержимое между скобками должно быть обработано в одной процедуре.
- Метод применим только к операторным грамматикам.

4.5.2.6. Операторные грамматики

Использование операторной грамматики – это еще один из методов разбора снизу–вверх. В этом методе ищется не предлог, а так называемая первичная фраза, то есть последовательность символов Tj, которые заключены между знаками «<·» и «·>».

Определение 1: **грамматика** является **операторной**, если не имеет правил формы U::=...VW..., где V и W нетерминалы (т.е. нахождение в правых частях правил двух РЯДОМ стоящих нетерминальных символов запрещено).

Определение 2: операторная грамматика является **грамматикой предшествия операторов**, если существует не более одного отношения между двумя любыми терминалами.

Определение 3: пусть имеем операторную грамматику, тогда

R·=S, тогда и только тогда, когда имеется правило U::=...RS... или U::=...RVS..., где V – нетерминал;

R<·S, тогда и только тогда, когда имеется правило U::=...RW..., W=>+S..., или W=>+VS...;

R·>S, тогда и только тогда, когда имеется правило U::=...WS..., W=>+...R, или W=>+...RV.

Пусть имеем простую грамматику арифметических выражений:

E ::= E+T| T

T ::= T*F| F

F ::= (E)| i

Матрица предшествия операторов для рассматриваемой грамматики выглядит так:

R\S	+	*	(	)	i
+	·>	<·	<·	·>	<·
*	·>	·>	<·	·>	<·
(	<·	<·	<·	·=	<·
)	·>	·>		·>	
i	·>	·>		·>	

Рис. 4.7. *Матрица для операторного предшествия*

Здесь R – первый оператор в строке, S – последующий оператор, i – обобщенный идентификатор.

Применяя такую матрицу, можно провести разбор с одним стеком, в отличие от других методов разбора снизу–вверх, где используется два стека.

Пример: Строка для разбора: i*(i+i)

Шаг	Стек	Отношение	R	Остаток строки
0	#	<·	I	*(i+i)#
1	#i	·>	*	(i+i)#
2	#N	<·	*	i+i)#
3	#N*	<·	(	+i)#
4	#N*(	<·	i	i)#
5	#N*(i	·>	+	i)#
6	#N*(N	<·	+	)#
7	#N*(N+	<·	i	)#
8	#N*(N+i	·>	)	#
9	#N*(N+N	·>	)	#
10	#N*(N	·=·	)	#
11	#N*(N)	·>	#	
12	#N*N	·>	#	
13	#N	·>	#	

1. Возникает резонный вопрос: а откуда взялся какой-то символ N и что он означает? Дело в том, что в данном методе нетерминальные символы не имеют весомого значения – нам не важно, какой именно символ – F, T или E стоит внутри правой части правила – важно, что это нетерминал. Поэтому N для нас представляет подстановку для ЛЮБОГО нетерминального символа.

2. Общая идея в нескольких словах – ищется конец первичной фразы. Первичная фраза заменяется на соответствующую ей строку с использованием нетерминала N, так как

все нетерминалы могут быть заменены этим символом. В конце разбора: N – начальный символ грамматики.

4.6. Контрольные вопросы по теме

1. В чем состоит основная задача парсера?
2. Какие существуют основные методы грамматического разбора?
3. Как грамматику расположить в оперативной памяти?
4. Что такое «возврат» при нисходящем грамматическом разборе? Причины его возникновения.
5. В чем состоит предназначение грамматики LL(1)?
6. Какие основные преимущества применения процедур рекурсивного спуска?
7. Необходимо ли наличие правил грамматики в памяти при использовании процедур рекурсивного спуска?
8. В чем состоит принцип факторизации и зачем он применяется?
9. В чем предназначение функций предшествия?
10. Будет ли разбор с помощью функций предшествия происходить быстрее, чем с помощью матрицы предшествия?
11. Почему при операторном предшествии все нетерминальные символы разбора заменяются единообразно?

5. ВНУТРЕННЯЯ ФОРМА ПРЕДСТАВЛЕНИЯ ИСХОДНОЙ ПРОГРАММЫ И СЕМАНТИКА ТАБЛИЦ

После синтаксического анализа некоторого предложения к нему применяется первая синтезирующая фаза компиляции, которая часто называется фазой семантических процедур. В результате ее работы отдельные предложения, которые сейчас существуют в виде последовательностей стандартных символов, преобразуются в так называемую «**Внутреннюю форму представления исходной программы**» (далее будем применять простое сокращение «**ВФП**»), более удобную для дальнейшей машинной обработки. Например, в языке JAVA используется форма представления, которую называют «байт-кодом».

ВФП – это фактически некоторый способ кодировки исходного текста, который совершенно не зависит от типа компьютера и, по большому счету, не зависит и от языка программирования. То, что ВФП является машинно-независимой формой представления, есть его самое большое преимущество. Фактически мы создаем некоторый псевдокод, который может быть выполнен специально для этой цели созданным интерпретатором, что мы и имеем, например, в случае с Java-интерпретатором.

Существуют повсеместно принятые «стандартные» типы ВФП, о некоторых из которых как наиболее популярных пойдет речь ниже.

Конечно, следует помнить, что выбор каждой частной ВФП зависит от исходного языка и от назначения компилятора. Например, в языке Фортран нет необходимости включать во внутреннюю форму исходной программы инструкцию DIMENSION, так как вся информация, содержащаяся в ней, попадет в таблицу символов и никакие команды генерироваться не будут (впрочем, в некоторых случаях такое включение может понадобиться). Описание INTEGER K также не встретится ни в одном из внутренних представлений, так как для него также не требуется генерация команд.

Следует также решить, насколько подробным должно быть начальное внутреннее представление. Включать ли во внутреннее представление операции преобразования значений из одного типа в другой или это делать позже? Например, цикл можно сразу представлять эквивалентной группой присваивания, переходов и условных инструкций, или его можно задавать с меньшей степенью детализации и транслировать позже.

5.1. Общие требования к внутренней форме представления

Все внутренние представления обычно содержат в себе два типа объектов – операторы и операнды. Различия лишь в том, как эти операторы и операнды соединяются. В дальнейшем мы будем использовать такие операторы, как +, –, /, ^, BR (Branch – переход) и т.д. Внутри компилятора они кодируются целыми числами. Таким образом, мы могли бы использовать число 4 вместо “+”, 5 вместо “–”, 6 вместо “/” и т.д.

Возможно, потребуется отличать операторы от операндов, если они могут встречаться в любом порядке. В таком случае операнд будет занимать две ячейки: в первой содержится целое число, несовпадающее ни с одним из чисел, представляющих операторы, а во второй – собственно операнд. (Иногда эти две компоненты упаковываются в одну ячейку.)

Операндами, с которыми мы будем иметь дело, являются простые имена переменных, процедур и т.д., константы, временные переменные, генерируемые самим компилятором, и переменные с индексами. Если все идентификаторы и константы хранить в одной общей таблице, то, за исключением переменных с индексами, каждый операнд может представляться указателем на соответствующий элемент в таблице символов. Если описания операндов находятся в нескольких таблицах, то в первой ячейке необходимо поместить признак типа операнда, например:

Простая переменная

1	**I**	**указатель на элемент таблицы**

Поле I – признак типа адресации

Временная переменная

2	**I**	**указатель на элемент таблицы временных**

Константа

3		**указатель на элемент таблицы констант**

5.2. Организация таблиц семантического анализа

Как мы видим, во всех полях представления операндов имеются ссылки (указатели) на некоторые таблицы. Что это за таблицы? Мы помним, что во время лексического анализа создавались таблицы разного назначения и в том числе таблица идентификаторов – важнейшая таблица компилятора, так как с ней работают практически все фазы компилятора.

Основной шаблон такой таблицы может быть представлен следующим образом:

Аргумент	**Дескриптор**

Аргументом является, естественно, сам идентификатор, а дескриптор – это поле произвольной длины и содержания, зависящее от предпочтений конструктора таблицы.

При организации такой таблицы сразу возникают некоторые проблемы.

Как определить длину поля аргумента? Ведь в разных языках программирования идентификаторам определяется самая разная длина – в Фортране это 6 символов, в Паскале – 64, из которых воспринимаются только 32 первых символа, и т.д. и т.п. Кроме всего прочего, программист может спокойно во всей программе использовать одно-двухсимвольные идентификаторы. То есть речь идет о том, как унифицировать длину поля аргумента.

Представим два способа записи переменных длин идентификаторов следующим образом.

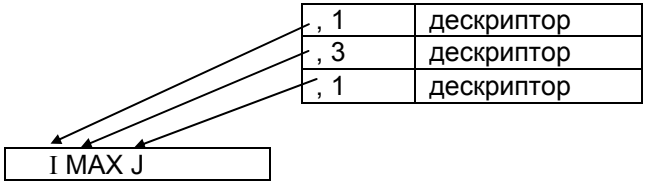


Рис. 5.1. *Идентификаторы в одномерном массиве*

В этом случае все представления идентификаторов располагаются в одномерном массиве, а в поле аргумента указывается индекс начала обозначения идентификатора и его длина.

Возможен еще один вариант, когда в одномерном массиве располагается и идентификатор, и его длина, а в поле аргумента остается только ссылка на начальную цифру длины.

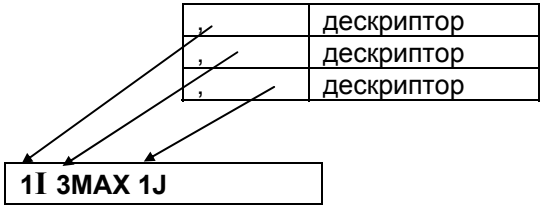


Рис. 5.2. *Идентификаторы в одномерном массиве*

Теперь обратимся к полю дескриптора. Какая информация об идентификаторе может быть записана в это поле? Конкретных требований на этот счет предъявить невозможно – ибо компиляторы бывают разные, языки – тоже, да и создатели компиляторов – тоже. Поэтому предлагается некоторый шаблон для заполнения этих полей в таблице. Естественно, что идентификатором могут обозначаться совершенно разные компоненты программы, и потому это задача конструктора – выделить главные характеристики той или иной компоненты, закодировать их и записать в поле дескриптора.

Пусть мы имеем идентификатор переменной величины (имя переменной). В принципе в дескриптор можно записать:

1. Тип переменной – REAL, INTEGER, BOOLEAN и т.д.
2. Длина переменной в байтах (точность представления).
3. Форма представления – простая переменная, индексированная, структура и т.д.
4. Адрес времени исполнения (runtime address).
5. Если это имя массива – то указать количество размерностей, количество индексов в каждой размерности. Если границы индексов переменны – указать и это.
6. Если это некоторая структура – указать на следующий компонент структуры.
7. Некоторые имена используются в списке формальных или фактических параметров, потому нужен тип соответствия.
8. Встретилось ли ранее в тексте программы описание этой величины?
9. Имеется в тексте программы предложение, инициализирующее эту величину?
10. Некоторые дескрипторы фиксируют использование данной переменной в качестве индекса цикла (внутри цикла не изменять).
11. Находится ли переменная в некоторых специфических областях памяти – например, в общей (COMMON) области или других – зависит от реализации языка.

Безусловно, все варианты здесь не могут быть изложены, и конструктор компилятора должен руководствоваться здравым смыслом и более точным соответствием требованиям языка.

Далее, некоторые идентификаторы могут представлять собою имена некоторых процедур (функций). В этом случае может быть полезна следующая информация в дескрипторе:

1. Является ли эта процедура внешней (EXTERNAL) по отношению к главной программной единице?
2. Является эта процедура функцией (тогда указать тип возвращаемого результата)?
3. Было ли обработано описание этой процедуры или нет (FORWARD в Паскале) или прототип в C/C++?
4. Является ли она рекурсивной?

5. Каковы ее формальные параметры? Формальные параметры должны быть по месту снабжены указателем на процедуру-хозяина.

В объектно-ориентированных языках возникают другие программные именованные объекты – классы, соответственно существуют их идентификаторы – и сопутствующая им информация, которая также должна быть записана в поля дескриптора.

5.3. Основные виды ВФП, применяемые в современных компиляторах

Рассмотрим описание и формат представления для наиболее известных видов ВФП.

5.3.1. Тетрады

Тетрада (ее можно назвать и «четверкой», по-английски – quadruple) представляет собой запись с четырьмя полями следующего формата:

Оператор	Операнд 1	Операнд 2	Результат

Рис. 5.3. *Формат тетрады (четверки)*

Адресация полей записи может быть заимствована из любого языка, работающего с записями (records) или структурами.

Такая форма представления является самой простой и поэтому популярной, в том числе и для демонстрации различных способов и приемов построения, так как она реализует так называемый *трехадресный код* (three-address code). С ее помощью можно представить и записать практически любую исполняемую конструкцию любого языка программирования высокого уровня. Исходная программа преобразуется в массив четверок, и каждая четверка адресуется соответствующим индексом массива. Например, для арифметического выражения A * B + C * D получается следующий массив четверок:

*, A, B, T1
*, C, D, T2
+, T1,T2,T3

Вид четверок в данном случае очень естественен, так как это представление арифметических операций. Однако не обязательно использовать все четыре поля, и поэтому внешний вид четверок может быть для разных действий самым различным. Надо отметить, что четверки являются наиболее эффективной формой внутреннего представления для логических выражений. Ниже в таблице представлены самые разнообразные виды четверок для наиболее часто используемых действий.

Оператор	**Операнды**	**Комментарий**
BR	i	Переход на i-ю тетраду
BZ [BP, BM]	i, P	Переход на i-ю тетраду, если значение, описанное в P, равно нулю [положительное, отрицательное]
BG [BL, BE]	i, P1, P2	Переход на i-ю тетраду, если значение, описанное в P1, больше [меньше, равно] значения, описанного в P2
BRL	P	Переход на тетраду, номер которой задан меткой (label) в P-м элементе таблицы символов
BMZ	i, P	Переход на i-ю тетраду, если значение, описанное в P, равно нулю или отрицательное
BPZ	i, P	Переход на i-ю тетраду, если значение, описанное в P, равно нулю или положительное
+ [–, *, /]	P1, P2, P3	Сложение [вычитание, умножение, деление] значений, описанных в P1 и P2, и запоминание результата по адресу, описанному в P3
:=	P1, , P3	Запись значения, описанного в P1, в ячейку, описанную в P3

CVRI	P1, , P3	Преобразование значения, описанного в P1, из типа REAL в тип INTEGER и запоминание результата в ячейке, описанной в P3
CVIR	P1, , P3	Преобразование значения, описанного в P1, из типа INTEGER в тип REAL и запоминание результата в ячейке, описанной в P3
BLOCK	–	Начало блока (begin)
BLCKEND	–	Конец блока (end)
BOUNDS	P1, P2	P1 и P2 описывают граничную пару массива
ADEC	P1	Массив описан в P1. Если размерность массива ***n***, то этой тетраде предшествуют ***n*** операторов BOUNDS, задающих n граничных пар

Рассматривая виды различных четверок, становится ясно, что не все четверки будут в дальнейшем служить базой для генерации кодов. Иными словами, не все четверки представляют собой «исполняемый» код. Типичными представителями таких четверок в этой таблице являются четверки BLOCK, BLCKEND, BOUNDS и ADEC. BLOCK и BLCKEND позволяют компилятору правильно определить границы блока инструкций или тел подпрограмм, а четверки BOUNDS и ADEC позволяют организовать правильное определение параметров массива и их дальнейшую адресацию.

В языке Алгол описание массива A записывается следующим образом:

ARRAY **A** [$L_1 : U_1, \ldots L_n : U_n$],

где Lower – нижняя граница индекса данной размерности, а Upper – верхняя. Например, в языке может быть разрешена такая запись:

A [–3:6,0:7]. В результате мы для работы с массивом нуждаемся в некотором операторе, позволяющем представить и имя массива, и описание любой размерности, и описание начального и конечного значения индекса в каждой размерности. Его можно представить в виде такой записи:

$L_1U_1 \ldots L_nU_n$ **A** ADEC

где ADEC (Array Declaration) – единственный оператор, представляющий собой обработку языкового описания массива. Он имеет переменное число операндов, зависящее от числа индексов. Операнд **A**, очевидно, будет адресом элемента таблицы идентификаторов для **A**. При вычислении ADEC из этого элемента таблицы извлекается информация о размерности массива **A** и, следовательно, о количестве операндов ADEC.

В конечном счете этот оператор представляется одной четверкой с двумя использующимися полями:

ADEC	P1	-	-

в которой определяется указатель на имя массива, а перед этой четверкой записывается столько операторов BOUNDS, сколько размерностей содержит наш массив, причем пары P1 и P2 определяют местонахождения граничных значений соответствующих индексов. При обработке этих четверок компилятор получает всю необходимую для работы с массивом информацию.

В качестве основного демонстрационного фрагмента будет использован фрагмент, написанный на языке Алгол.

```
BEGIN INTEGER K;
ARRAY A[1:I-J];
K:=0;
L: IF I>J    THEN K:=K+A[I—J]*6
ELSE BEGIN I: =I+1; I:=I+l; GO TO L END
END
```

Пусть читатель не смущается тем, что он может не знать этого языка, фрагмент этот вполне понятен для студента 3-го курса, и он хорош тем, что здесь встречаются сразу несколько достаточно сложных конструкций, которые на примере Алгола легко объяснимы. Присутствие двух подряд предложений присваивания I:=I+1 не является ошибкой или опечаткой.

Применяя сконструированные таким образом четверки для нашего учебного фрагмента, получаем массив из 18-ти четверок:

```
(1)     BLOCK              (10)    + K, T4, T5
(2)     - I, J, T1         (11)    := T5, , K
(3)     BOUNDS 1,T1        (12)    BR 18
(4)     ADEC A             (13)    + I, 1, T6
(5)     := 0,,K            (14)    := T6, , I
(6)     - I, J, T2         (15)    + I, 1, T7
(7)     BMZ 13, T2         (16)    := T7, , I
(8)     — I, J, T3         (17)    BRL L
(9)     * A [T3],6,T4      (18)    BLCKEND
```

5.3.2. Четверки для построения логических выражений

Для логических выражений можно использовать тетрады следующих видов:

1) := , значение , , операнд

используется для присваивания ведущему (результирующему) операнду значения: 1 – истина (TRUE), 0 – ложь (FALSE). Ведущий операнд содержит результирующее значение всего логического выражения и может меняться в зависимости от его истинности или ложности. В начале вычислений этому операнду присваивается значение 1 и далее идут расчеты в соответствии с операторами и значениями операндов, и если в результате выясняется, что выражение истинно, то значение этого операнда так и остается равным 1. В противном случае значение этого операнда устанавливается в 0.

2) B, операнд , nBNZ , nBZ

является основным видом тетрады, который используется для вычислений значения логического выражения. Здесь:

B – оператор четверки, обозначающий логическую четверку;

операнд – имя или ссылка на некоторую логическую величину;

nBNZ – адрес тетрады, на которую будет осуществлен переход в случае, если значение операнда истинно – не ноль;

nBZ – соответственно адрес тетрады, на которую будет осуществлен переход в случае, если значение операнда ложно (0).

Пример:

Выражение `a AND (b OR c)` (величины a, b, c – логические)

```
0    :=    1     -     X
1     B    a     2     4
2     B    b     5     3
3     B    c     5     4
4    :=    0     -     X
5 ...................
```

Результат вычисления формируется в ячейке **X**, и первой же четверкой он устанавливается в TRUE, а затем происходит само вычисление значения логического выражения, и при необходимости изменения значения **X** мы попадаем на четвертую тетраду. Тетрада с номером 5 представляет собой продолжение массива четверок.

Эта запись имеет своим достоинством то, что она компактна и легко оптимизируема (действительно, очень удобно осуществлять проход по массиву таких тетрад и просматривать все поля, например nBNZ), а недостатком – что далека от машинного представления.

Поэтому существует еще один вид «логических» четверок:

оператор, адрес перехода, операнд, 0,

где оператор – один из операторов: BNZ, BZ – предложенные стандартно (здесь можно применять и оператор BR – что более оптимизирует программу).

Этот вид четверок более тесно связан с машинным представлением, и по нему уже совсем несложно генерировать машинные команды, в отличие от предыдущего примера, где возникает проблема раскрытия четверок до машинных команд и, как последствие, пересчет адресов перехода и другие проблемы технического характера.

Тот же пример: выражение `a AND (b OR c)`

```
0    :=    1     -     X
1    BZ    4     a     -
2    BNZ   5     b     -
3    BNZ   5     c     -
4    :=    0     -     X
5 ...................
```

Подобные способы внутреннего представления имеют одну очень важную характерную особенность – такая организация вычисления исполняемым кодом значения логического выражения позволяет значительно сократить длительность самого процесса вычисления, согласуясь с правилами из таблиц истинности для каждого из операторов, что дает возможность прекратить вычисления сразу как только становится известным окончательный результат. Фактически при построении четверок мы производим оптимизацию логических выражений – выполнение как можно меньшего количества операций. То есть в выражении A AND (B OR NOT C), при A = FALSE, дальнейшие операции не выполняются (так называемые «ленивые вычисления»).

Рассмотрим предыдущий пример более подробно:

– если обратить внимание на строку с номером 1, то видно, что, если аргумент *a* ложен, то сразу выполняется переход на строку с номером 4, где в ведущий операнд выражения **X** заносится значение 0. В противном случае – процесс вычисления продолжается дальше по порядку следования операторов;

– если обратить внимание на строку с номером 2, то видно, что если аргумент *b* истинен, то сразу выполняется переход на строку с номером 5 и результатом вычисления выражения будет истина (так как ведущий операнд **X** остался равным 1). В противном случае процесс вычисления продолжается дальше по порядку следования операторов.

Таблицы истинности

a	b	a AND b	a OR b	a XOR b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

	not a
	1
	0

Здесь AND, OR, XOR – бинарные (двуместные) операции, а NOT – одноместная.

Как видно из таблицы, в отличие от остальных операций, для оператора XOR значение выражения A XOR B не может быть определено по одному из операндов, а только по двум – это делает специфичным вычисления выражений с таким оператором.

5.3.3. Тройки

К недостаткам тетрад следует отнести большое количество временных переменных, которые приходится применять для хранения как промежуточных, так и конечных результатов. Эта проблема полностью отпадает при использовании троек (триад, по-английски – triple). Триада имеет следующую форму – <оператор><операнд1><операнд2> – то есть это запись с тремя полями, при этом каждая тройка имеет обязательный номер (индекс) в массиве представления.

Оператор	Операнд 1	Операнд 2

***Рис. 5.4**. Формат триады (тройки)*

С помощью троек наш фрагмент представляется следующим образом:

```
(1) BLOCK                (10) + K, (9)
(2) - I, J               (11) := (10), K
(3) BOUNDS 1, (2)        (12) BR (18)
(4) ADEC A               (13)+ I, 1
(5) := 0, K              (14) :=,(13),I
(6) - I,J                (15)+ I, 1
(7) BMZ (13), (6)        (16) := (15), I
(8) — I, J               (17) BRL L
(9) * A[(8)], (6)        (18) BLCKEND
```

В триаде нет поля для результата. Если позднее какой-то операнд окажется результатом данной операции, то он будет непосредственно на нее ссылаться. Например, выражение `A+B*C` будет представлено следующим образом:

```
(1) * B, C
(2) + A, (1)
```

Здесь (1) – это ссылка на результат первой триады, а не константа, равная 1. Выражение `1+B*C` будет записано так:

```
(1) * B, C
(2) + 1, (1)
```

Триады любого арифметического выражения можно также рассматривать как прямое представление дерева. *Последняя* триада соответствует корню дерева. Каждая i-я триада соответствует поддереву, оператор триады является корнем поддерева, а каждый операнд – либо именем переменной, которая соответствует концевому узлу, либо номером триады, описывающей поддерево.

От того, как рассматриваются триады (как последовательность операций в порядке их выполнения или как дерево), существенным образом зависит генерируемый объектный код.

Пример построения дерева для выражения `A*B+C-D*E`:

```
(1)    (* A,   B)
(2)    (+ (1), C)
(3)    (* D,   E)
(4)    (- (2), (3))
```

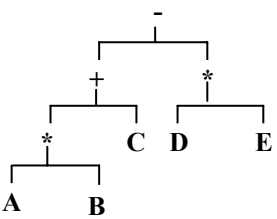


Рис. 5.5. *Пример представления дерева триадами*

Тройки являются очень важным видом ВФП. Во-первых, с их помощью можно получить существенно более эффективный объектный код – примерно на 10–20%, по сравнению с четверками. Правда, за большее время компиляции. Во-вторых, тройки в их текстовом представлении являются самой эффективной основой для различных типов машинно-независимой оптимизации, что делает их во многих случаях основным типом ВФП для оптимизирующих компиляторов.

5.3.4. Косвенные тройки

Во время оптимизации объектного кода обычно требуется какие-то операции исходной программы исключить, какие-то переставить на другое место. С тетрадами это делается легко, а с триадами сложнее, так как они часто ссылаются друг на друга.

Для решения этой проблемы заведем две таблицы. В одной таблице (ТРИАДЫ) будем хранить триады, а в другой таблице (ОПЕРАЦИИ) – последовательность ссылок на триады в том порядке, в каком они должны выполняться. Например, инструкции A:=B*C и B:=B*C представляются в следующем виде:

ОПЕРАЦИИ	ТРИАДЫ
1 (1) 2 (2) 3 (1) 4 (3)	(1) * B, C (2) := (1), A (3) := (1), B

Теперь мы можем переставлять или исключать операции в таблице ОПЕРАЦИИ, не изменяя сами триады и ссылки на них. Заметим, что одинаковые триады в таблице ТРИАДЫ не повторяются. Две триады считаются одинаковыми, если у них совпадают все три компоненты. Операнды, соответствующие двум разным переменным B, описанным в разных блоках, конечно, будут разными. Обычно в исходной программе много одинаковых триад, особенно если встречаются переменные с индексами, и поэтому общее количество триад меньше количества операций.

5.3.5. Обратная польская запись

Обратная польская запись – в дальнейшем ОПЗ – представляет собой способ записи, изобретенный польским математиком Лукашевичем, часто также называемая постфиксной записью для выражений, содержащих операторы и операнды. Так, запись **a + b * c** трансформируется в **bc*a+.** В этой записи двуместный оператор записывается после операндов, к которым он применяется. В нашей обычной практике мы пользуемся инфиксной записью, когда оператор стоит между операндами.

По многим характеристикам ОПЗ является очень удобным типом ВФП и часто применяется для представления арифметических и логических выражений. В данном конспекте мы не будем подробно обсуждать эту форму ВФП, так как ее описание можно легко найти во многих первоисточниках.

5.3.6. Представление программы в виде графа

Часто бывает удобно разбить программу на «линейные участки» и отдельно иметь «граф программы», описывающий, как эти участки связаны между собой. Линейный участок – это последовательность операций с одним входом и одним выходом (им соответствуют первая и последняя операции). Более того, все операции выполняются последовательно и внутри участка нет переходов. Далее в качестве примера для нашего демонстрационного сегмента программы приведены линейные участки в виде тетрад (с тем же успехом можно было использовать косвенные триады). Заметим, что в операциях переходов в качестве операндов указываются номера линейных участков, а не номера тетрад. Тетрада BMZ во 2-м линейном участке означает следующее: если значение T2 меньше или равно нулю, то происходит переход на 4-й линейный участок, в противном случае – переход на 3-й участок. В общем случае последняя операция может быть переходом на любой линейный участок. Заметим также, если участок имеет один преемник, то в конце нет необходимости явно указывать переход на него, так как этот переход неявно указан в графе программы.

```
1       BLOCK              3       - I, J, T3
        - I, J, T1                 * A[T3], 6, T4
        BOUNDS 1,T1                + K, T4, T5
        ADEC A                     :=T5,, K
        := 0, , K          4       + I, 1, T6
2       - I, J, T2                 :=T6,, I
        BMZ 4,3, T2                + I, 1, T7
                                   :=T7,, I
                           5       BLCKEND
```

$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Рис. 5.6. *Пример представления программы в виде графа*

Граф программы содержит для каждого линейного участка список непосредственных преемников (к которым от него есть переход) и непосредственных предшественников (от которых к нему есть переход). Граф программы изображается в виде матрицы **M**, для которой **M**[i, j]=1 в том и только в том случае, если j-й линейный участок является преемником i-го линейного участка. На практике для представления графа программы лучше подходит списочная структура, так как, возможно, потребуется хранить больше информации о каждом линейном участке, а также создавать или уничтожать участки в процессе оптимизации.

Следует отметить особую важность и значимость для компилятора уметь представить программу в графовом виде, потому что это сейчас является во многих случаях необходимым условием для взаимодействия архитектуры процессора и компилятора при создании высокоэффективных параллельных вычислений, при которых отдельные линейные участки программы могут исполняться одновременно.

5.3.7. Другие формы представления ВФП

При создании компиляторов для языка Паскаль использовалась другая форма внутреннего представления исходной программы, которая носит название P-код. Этот код является промежуточным кодом на основе стека, созданным специально для реализации языка Pascal и широко используемым для этой цели. Каждая инструкция P-кода имеет следующий формат, по сути являясь модификацией троек.

```
F P Q
```

Здесь F – код функции, а P или (иногда – и) Q могут отсутствовать в зависимости от конкретного кода. При наличии этих параметров P может применяться для определения уровня статического блока, а Q – для определения офсета внутри фрейма или промежуточного операнда (например, константы). **P-код** – концепция аппаратно-независимого исполняемого кода в программировании, введенная в начале

1970-х годов, в том числе с участием Никлауса Вирта. Этот термин в основном применяется для обозначения реализации виртуальной машины для языка Паскаль, но иногда обобщается на виртуальные машины вообще (например, виртуальная Java-машина, байт-код MATLAB).

Более популярная сейчас форма ВФП – это байт-код. Байт-код представляет собой промежуточный язык для Java Virtual Machine (JVM). Он, подобно P-коду для языка Pascal, основан на использовании стека. Отметим, что Java Virtual Machine разрабатывалась, чтобы реализации Java были:

- эффективными;
- защищенными;
- переносимыми.

Для каждого *класса* инструкции байт-кода находятся в *классификационном файле Java* (Java class file). В каждом файле содержится виртуальный машинный код для используемых классом методов (функций/процедур), информация таблицы символов (*набор констант* в Java), соединений с суперклассами и т.д. Для эффективной работы файл имеет двоичный формат, но для удобства просмотра его можно преобразовать в символьную форму. Существенной особенностью реализаций Java является наличие *верификатора классификационного файла* (class file verifier), который, помимо всего остального, контролирует, чтобы файл, поступивший с ненадежного источника, не вызвал сбой в работе интерпретатора, оставив его в неопределенном состоянии, или аварию хоста. В частности, верификатор байт-кода используется для проверки байт-кодов внутри методов на предмет:

- наличия команд ветвления, обращающихся к неправильным адресам;
- ошибок типов в кодах инструкций;
- некорректного управления стеком по отношению к условиям переполнения и опустошения;
- методов, вызываемых с неправильным числом или типом аргументов.

Существует более 160 различных инструкций байт-кода, многие из них отличаются только типами операндов. Для верификации важным является хранение информации о типах в байт-коде, так как множество имеющихся инструкций по-разному поддерживают различные типы данных. Основные типы инструкций байт-кода можно выделить в такие группы:

- работа со стеками;
- выполнение арифметических операций;
- оперирование объектами и массивами;
- поток управления;
- вызов методов;
- обработка исключительных ситуаций и параллельная работа.

5.4. Контрольные вопросы по теме

1. Как решается вопрос об унификации длин идентификаторов для хранения в таблице символов?
2. Какая информация о б идентификаторе может храниться а таблице символов?
3. Что такое четверка (тетрада)?
4. Каково представление программы в виде четверок?
5. Как описания массивов формируют четверки и почему?
6. Почему четверки удобны для представления логических выражений?
7. Тройки (триады) и их главные отличия от четверок.
8. Зачем нужны косвенные тройки?
9. Для каких вычислений удобно представление программы в виде графа?
10. Какие другие формы ВФП вы знаете?

6. Введение в семантические процедуры

Итак, мы рассмотрели различные виды внутренней формы представления исходной программы. Теперь наша задача – спроектировать процедуры, которые умеют создавать в ходе компиляции эти формы представления. Эти процедуры называются семантическими, потому что в результате их работы получается промежуточная форма представления программы, полностью адекватная смыслу, который был заложен в исходную программу на языке высокого уровня.

Обычно семантические процедуры интегрируют с синтаксическим анализом. Как только парсер разобрал некоторую конструкцию языка и признал ее верной, то есть соответствующей грамматике, уже может быть готова и соответствующая ВФП. Это наиболее популярное решение, хотя и не совсем рациональное – ведь если в конце инструкции будет обнаружена ошибка, это означает, что созданные ранее для начальной части предложения тройки или четверки придется удалять.

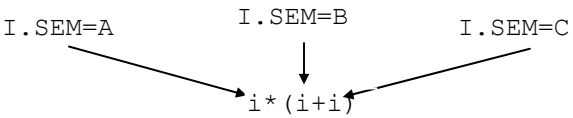
Рассмотрим пример построения фазы семантического анализа на примере грамматики для арифметических выражений:

```
Z::= E
E::= T|E+T|E-T|-T
T::= F|T*F|T/F
F::= i |(E)
```

Наиболее просто продемонстрировать работу семантических процедур при грамматическом разборе «снизу–вверх». Пусть у нас есть арифметическое выражение A*(B+C). На вход синтаксического анализатора оно поступает в виде строки стандартных символов i*(i+i), в которой вместо конкретных имен переменных используется ”i“ – так называемый обобщенный идентификатор. Именно имя переменной и представляет для нас семантику, ведь «по смыслу» мы должны сложить B с C, дать результату какое-то временное имя и затем его умножить на A. Это значит, что при самой первой замене (свертке) i на F надо семантику,

связанную с данным i, передать в семантику F. Обычно это записывается так:

F.SEM=I.SEM



Затем надо при каждом применении правила во время свертки передавать эту семантику «вверх по дереву», при необходимости генерируя новые временные имена для хранения результата и также необходимые четверки, если правило содержит арифметические операции.

В данном случае мы используем уже известный нам способ представления грамматики как атрибутивной грамматики. Именно информационность такой формы является наиболее привлекательной при реализации семантического анализа.

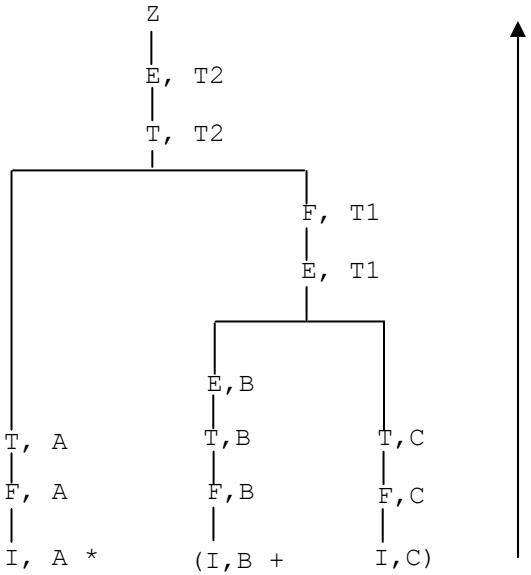


Рис. 6.1. *Проводка семантики снизу–вверх*

На этом дереве продемонстрирован весь процесс «проводки» семантики при грамматическом разборе снизу–

вверх для переменных величин арифметического выражения A*(B+C).

В процессе семантического анализа будут сгенерированы две четверки:

```
+, B,C,T1
*, A,T1,T2
```

Генерация четверки и запись ее в массив осуществляется вызовом после свертки специальной процедуры Make(W, X, Y, Z), где W – параметр для знака операции четверки, а X, Y, Z – параметры для двух операндов и результата.

Правило	Семантическая процедура
(1) Z::=E	Z.SEM :=E.SEM.
(2) E::=T	E.SEM :=T.SEM.
(3) E_1::=E_2+T	i := i+1; E_1.SEM:= Ti;
	Make("+",E_2SEM,T.SEM, E_1.SEM).
(4) E_1::=E_2-T	i :=i+1; E_1.SEM:=Ti;
	Make("-",E_2.SEM,T.SEM,E_1.SEM).
(5) E::= -T	i :=i+1; E.SEM:=Ti;
	Make ("@", 0, T.SEM, E.SEM).
(6) T::=F	T.SEM :=F.SEM.
(7) T_1::=T_2*F	i := i+1; T_1.SEM:= Ti.
	Make ("*", T_2.SEM, F.SEM, T_1.SEM).
(8) T_1::=T_2/F	i := i+1; T_1.SEM:=Ti;
	Make ("/", T_2.SEM, F.SEM, T_1.SEM).
(9) F::=I	F.SEM :=I.SEM.
(10)F::=(E)	F.SEM :=E.SEM.

Знак @ здесь применяется для обозначения одноместного минуса. Счетчик i обеспечивает изменение имен временных переменных. Обозначения вида E_1, E_2 вводятся для возможности различения нетерминала в левой и правой частях правил.

6.1. Семантическая обработка при рекурсивном спуске

Чтобы использовать рекурсивные процедуры для грамматического разбора и трансляции предложений грамматики для арифметических выражений, нам прежде всего придется переписать грамматику следующим образом:

```
Z ::= E
E ::= [-]T{(+|-) T}
T ::=F{(*|/)F)
F ::=I|(E)
```

используя уже известный прием факторизации.

Затем напишем следующие четыре рекурсивные процедуры:

```
PROCEDURE Z;                          Для правила Z ::= E.
BEGIN E END;
PROCEDURE E;                          Для правил E ::= [-]T{(+|-)T}.
BEGIN                                 Проверка на первый одноместный
                                      минус.
IF NS = '-' THEN    SCAN;T;           Получение терма. Итеративный
                                      поиск вхождений +T или -T.
WHILE NS = '+' OR   NS='-'
DO BEGIN SCAN; T
END END;
PROCEDURE T;                          Для правил T ::= F{(*|/)F}.
BEGIN F;                              Сначала разбор множителя.
WHILE NS ='*' OR                      Затем итеративный поиск
NS ='/' DO                            вхождений *F или /F.
BEGIN SCAN; F END   END;
PROCEDURE F;                          Для правил F ::= I|(E) .
BEGIN
IF NS = 'I' THEN    SCAN;             Проверка на идентификатор.
ELSE                                  Если не I, то множитель должен
                                      начинаться с открывающей
                                      скобки; после нее должны
                                      следовать E и закрывающая
                                      скобка.
IF NS ≠'('THEN ERROR
       ELSE BEGIN SCAN; E;
IF NS ≠ ')' THEN ERROR ELSE
SCAN END END;
```

Так как в этих процедурах отсутствует какая-либо семантическая обработка, дополним каждую процедуру инструкциями для необходимой семантической обработки символов. Будем генерировать тетрады. Проиллюстрируем эту обработку на примере процедур T и F, оставив возможность читателю сделать то же самое для двух оставшихся процедур E и Z.

Заметим, что синтаксическое дерево и семантический стек при рекурсивном спуске явно не заданы. Вместо этого в каждой процедуре используются локальные переменные, а формальные параметры позволяют передавать семантическую информацию.

Семантикой для любого нетерминала Z, E, T или F является имя переменной из исходной программы или имя временной переменной. Это имя необходимо связать с нетерминалом, как только закончен разбор фразы для этого нетерминала. В нашем случае это выход из некоторой процедуры «вверх». Для этого в каждой из процедур предусматривается параметр X типа STRING – строчный, и после выполнения процедуры имя соответствующей переменной возвращается в X.

Ниже приводится измененная процедура F(X). Программируя ее, мы исходили из предположения, что процедура SCAN, если встретился идентификатор, поместит его признак ‘I’ в NS, а сам идентификатор в NSSEM. Заметим, что в процедуре имеется обращение к E(X), которая в X вернет имя переменной для фразы E. То же самое имя используется и для F.

```
PROCEDURE F(X); STRING X;
BEGIN
IF NS = 'I' THEN
BEGIN X := NSSEM; SCAN
END
ELSE
IF NS  ≠ '(' THEN ERROR
ELSE
BEGIN SCAN; E(X);
IF NS  ≠ ')' THEN ERROR
ELSE SCAN
END END
```

Семантика F (непосредственное имя переменной) возвращается в X.

Если идентификатор, то его имя SCAN поместит в NSSEM.

Если (E), то проверка скобок и затем разбор E. Имя, связанное с E, будет также именем, связанным с F.

Рассмотрим теперь процедуру T, в которой генерируются тетрады для * и /. Именно в ней надо «впервые» сгенерировать четверку для производства операции и сохранения ее результата в новой временной переменной. Сначала делается разбор множителя, и его имя заносится в локальную переменную Y. Затем мы сканируем операцию * (или /) и второй операнд, после чего генерируется тетрада. Заметим, что после каждой генерации тетрады в Y заносится имя переменной, содержащей результат операции. Этот процесс повторяется до тех пор, пока не будут обработаны все операции * и /.

```
PROCEDURE T(X); STRING X;
```
В X возвращается имя переменной для фразы T.

```
BEGIN STRING Y, Z, OP;
F(Y); OP :=NS;
WHILE OP='*'OR OP='/'DO
```
Разбор первого множителя; его имя заносится в локальную переменную Y.

```
    BEGIN SCAN;
    F(Z); j:=j+l;
```
На каждой операции вызывается F для разбора следующего множителя, генерируется имя временной переменной, генерируется тетрада. Имя результата есть Tj. Переменная j – глобальная.

```
Make (OP, Y, Z, Tj);
```
Генерация тетрады.

```
Y := Tj; OP := NS;
      END X := Y; END
```
Пересылка имени результата из Y в возвращаемый параметр X.

Итак, после интеграции семантической обработки в конкретный вид синтаксического анализатора компилятор становится способным генерировать внутреннюю форму представления, которая после машинно-независимой оптимизации станет базой для дальнейшей генерации уже непосредственно ассемблерного кода.

6.2. Семантическая обработка инструкций переходов

Рассмотрим эту семантику на примере широко известной базовой инструкции типа GO TO m. Для того чтобы дать полное грамматическое описание этой инструкции, применяются несколько грамматических правил. В грамматике принято метку, к которой следует перейти, обозначать совместно с инструкцией <инстр2>, переход на которую помечен этой меткой.

<инстр1> ::= <определение метки><инстр2>

<определение метки> ::= I:

Например: A**:** B := 1.5;

В данном случае в качестве метки мы рассматриваем обобщенный идентификатор метки I. Обработка нетерминала <определение метки> производится специальной процедурой примерно такого содержания:

`LOOKUPDEC(I.NAME,P);`	Просматриваем в таблице идентификаторов имя, связанное с I (имя метки). P – возвращаемый указатель (индекс)
`IF P=0 THEN BEGIN`	Если идентификатор метки отсутствует, то
`INSERT(I.NAME,P);`	вставляем его в таблицу.
`P.TYPE:=LABEL END;`	В поле типа записываем тип метки
`ELSE BEGIN`	Если идентификатор имеется в таблице,
`CHECKTYPE(P,LABEL);`	проверяем его на тип метки, и если эта метка уже описана, то фиксируем
`IF P.DECLARED=1`	
`THEN ERR(n) END;`	ошибку повторного описания (№ n)
`P.DECLARED:=1;`	Если ранее не описывалась – фиксируем
`P.ADDRESS:=NXQUAD;`	В поле адреса пишем номер следующей четверки

Теперь рассмотрим само применение метки: **<инстр> ::= GOTO I**

Семантическая процедура для обработки:

`LOOKUP(I.NAME,P);`	Поиск метки в таблице. P – возвращаемый указатель (индекс).
`IF P=0 THEN BEGIN`	Если идентификатор метки отсутствует, то вставляем его в таблицу.
`INSERT(I.NAME,P);`	
`P.TYPE := LABEL;`	В поле типа записываем тип метки.
`P.DECLARED :=0;`	Определение метки ранее не встретилось.
`END;`	
`ELSE`	Если идентификатор имеется в таблице,
`CHECKTYPE(P,LABEL);`	проверяем его на тип метки.
`ENTER(BRL,P,0,0)`	Записываем текст четверки в массив.

6.3. Проблема «перехода вперед»

Вместо применения оператора BRL (переход по метке), когда метка указывается по ее указателю P в таблице идентификаторов, чаще используют оператор BR (безусловный переход) с указанием непосредственного номера той тетрады, на которую надо перейти. При переходе «назад» по тексту программы здесь проблем не возникает – так как номер такой тетрады уже существует, она уже была сгенерирована. А вот при переходе «вперед» по тексту программы этого сделать нельзя, так как тетрады еще не сгенерированы. Для решения этой проблемы применяется идея связанного списка.

Рассмотрим такой вариант программы:

```
        GOTO M;          Генерируем (10) BR 0,0,0
        ……
        GOTO M;          Генерируем (20) BR 10,0,0
        ……
        GOTO M;          Генерируем (40) BR 20,0,0
        .....
M :     Инструкция;      (60)
```

Здесь слева от тетрады проставлены номера гипотетической четверки, которые будут так нумероваться при дальнейшей генерации. Нужная нам четверка будет иметь номер 60, и именно переход на нее должен быть проставлен во всех трех тетрадах.

При генерации четверок соответствующие поля дескриптора для метки M будут изменяться следующим образом

ID	Address (номер четверки)	Type	Declared
M	10	Label	0
M	20	Label	0
M	40	Label	0
M	60	Label	1

6.4. Контрольные вопросы по теме «Семантические процедуры»

1. В чем состоит основной результат работы семантических процедур?
2. Что такое "проблема перехода вперед"?
3. Какой вид ВФП наиболее часто применяется для логических выражений и почему?
4. Что такое «семантика нетерминального символа»?
5. Предложите ВФП для вызова процедуры или функции.
6. Каковы известные методы организации взаимодействия синтаксиса и семантики?
7. В каком случае семантическая обработка занимает более длительное время – при Top-down или Down-top методах грамматического разбора?
8. Как могут адресоваться поля троек и четверок?
9. Могут ли некоторые объекты ВФП быть «неисполняемой» формой представления и что это значит?

7. МАШИННО-НЕЗАВИСИМАЯ ОПТИМИЗАЦИЯ

Каковы основные цели этого процесса? В основном целью любой оптимизации считается уменьшение размеров объектного (финального) кода, что позволяет экономить оперативную память, и увеличение быстродействия машинного кода. Это две главные задачи. Как правило, первый этап автоматизированной оптимизации выполняется на уровне ВФП.

7.1. Классификация методов оптимизации

По уровню представления программы:

– на уровне текста исходной программы;

– машинно-независимая (на уровне машинно-независимого промежуточного представления – ВФП);

– машинно-зависимая (на уровне машинного языка, учитывая архитектурные особенности Hardware).

По размеру фрагмента оптимизации:

– локальная (линейный участок, группа операторов);

– глобальная (процедура целиком);

– межпроцедурная.

Далее будут рассмотрены приемы и алгоритмы оптимизации на базе ВФП, в частности триад (троек). Очевидно, что такая оптимизация является машинно-независимой оптимизацией (МНО).

7.2. Основные направления оптимизации ВФП

- Выполнение операций над операндами, значения которых уже известны на этапе компиляции
- Удаление пустого кода
- Удаление избыточных операций (например, вынесение общего множителя за скобки)
- Вынесение из тела цикла операций, чьи операнды не изменяются в цикле
- Уменьшение мощности операций

Все направления МНО на уровне ВФП выполняются в пределах так называемого линейного блока – последовательности инструкций языка, выполняющейся последовательно и имеющей один вход и один выход и не содержащей команд передачи управления. Оптимизация ВФП проводится в пределах линейного блока.

Рассмотрим первое направление МНО.

7.2.1. Выполнение операций над операндами, значения которых уже известны на этапе компиляции

Для демонстрации изберем некоторый участок кода, назначение которого чисто демонстрационное – очевидно, что «нормальный» программист такого в программе в действительности никогда не напишет:

```
I = 1+1
I = 3
B = 6.2+ I
```

Соответствующие тройки для данного линейного участка будут иметь следующий вид:

```
(1)     + 1, 1
(2)     = (1), I
(3)     = 3, I
(4)     CVIR I
(5)     + 6.2, (4)
(6)     = (5), B
```

Зададим некоторые установочные параметры, переменные и понятия, необходимые для алгоритма этого вида оптимизации:

A – переменная;

K – текущее значение переменной;

T – рабочая таблица, во время оптимизации заполняется упорядоченными парами (A,K).

Удаленные тройки будут замещены тройками вида [C K, 0], которые будут опущены при кодогенерации.

C – оператор-константа;

K – значение константы.

1. Если операнд – это переменная из таблицы T, заменить этот операнд на его значение.

2. Если операнд – это адрес тройки вида [C K, 0], заменить этот операнд на его значение.

3. Если оба операнда – константы, то заменить тройку на [C K, 0], где K – результат текущей операции.

4. Если тройка имеет вид A=B, то:

- Если B = const, добавить переменную A со значением B в T.
- Если B ≠ const, удалить переменную A со значением B из T.

Процесс оптимизации – все шесть шагов (по количеству троек) – показан ниже.

```
(1)C 2, 0          C 2, 0          C 2, 0
(2)= (1), I        = 2, I          = 2, I
(3)= 3, I          = 3, I          = 3, I
(4)CVIR I          CVIR I          CVIR I
(5)+ 6.2,          + 6.2, (4)      + 6.2, (4)
(4)                = (5), B        = (5), B
(6)= (5), B        T(I,2)          T(I,3)
   C 2, 0          C 2, 0          C 2, 0
   = 2, I          = 2, I          = 2, I
   = 3, I          = 3, I          = 3, I
   C 3.0, 0        C 3.0, 0        C 3.0,0
   + 6.2,          C 9.2, 0        C 9.2, 0
   (4)             = (5), B        = 9.2, B
   = (5), B        T(I,3)          T(I,3)
   T(I,3)                          T(B,9.2)
```

В дальнейшем тройки вида [C K, 0] не участвуют в кодогенерации!

Этот метод, называемый также методом «свертки констант» (константная арифметика или распространение констант), сводит выражения, которые содержат константные данные, к возможно простейшей форме. Константные данные обычно используются в программе либо непосредственно (как в случае чисел или цифр), либо косвенно (как в случае объявленных манифестных констант). Например, свертка констант сводит следующие инструкции:

```
#define TWO 2
a = 1 + TWO;
```

к эквивалентной форме,

```
a = 3;
```

во время компиляции, благодаря чему удаляются ненужные арифметические операции из стадии выполнения программы. В компиляторах Си сворачивание констант применяют как к целым константам, так и к константам с плавающей точкой.

7.2.2. Удаление избыточных операций

Это следующее очень часто применяющееся направление оптимизации.

Избыточность может появляться в результате небрежности или невнимательности программиста, его поспешности и т.д. и т.п.

Продемонстрируем такую избыточность.

D=D+C*B	A=D+C*B	C=D+C*B
(1) *C,B	(4) *C,B	(7) *C,B
(2) +D,(1)	(5) +D,(4)	(8) +D,(7)
(3) =(2),D	(6) =(5),A	(9) =(8),C

Очевидна избыточность двух выражений C*B.

Однако существуют случаи, когда такую избыточность программист просто физически не может увидеть, например, при работе с индексированными переменными. Пусть имеем обычное присваивание для индексированных переменных – X(I,J)= X(I,J+1). Тройки будут выглядеть следующим образом:

(1) *i,d2	(3) *i,d2	(5) +(4),1
(2) +(1),,j	(4) +(3), j	(6) = X((2)),X((5))

Очевидно, что для программиста эти операции недоступны – вычисление адресов операндов по их индексам производит программа во время исполнения, и определить избыточность, которая тут очевидна, он не может.

Таким образом, i-я операция в линейном блоке является избыточной, если ранее имелась идентичная операция j и ни одна операция между ними не изменяет значение переменных из операции j .

Введем новую тройку:

– SAME j, 0 (j – номер тройки, из-за которой наша тройка является избыточной, SAME – такая же).

- Введем коэффициент зависимости:

– dep(A) – коэффициент для переменной;

– dep(i) – коэффициент для тройки.

Установка коэффициента зависимости

1. Начальное значение dep(A) = 0.
2. Если, после обработки i-й тройки, она присваивает значение переменной i, то dep(A)=i.
3. При обработке i-й тройки ее коэффициент устанавливается в 1+max{dep(OP1),dep(OP2)}.

- Если тройка j идентична i, причем j<i, то i-я тройка избыточна тогда, когда: dep(i)=dep(j).

Алгоритм удаления избыточности

1. Если операнд тройки адресуется к тройке в форме SAME j, 0 , то заменить этот операнд на j.
2. Вычислить dep(i) по формуле.
3. Если существует идентичная j (j<i) тройка и dep(i)=dep(j), то заменить тройку i на SAME j, 0.
4. Если i-я тройка присваивает значение переменной D, то установить dep(D)=i.

Пример

Тройки	**Коэффициенты зависимости** A, B, C ,D	**dep(i)**	**Результат**
(1) *C,B	0 0 0 0	1	*C,B
(2) +D,(1)	0 0 0 0	2	+D,(1)
(3) =(2),D	0 0 0 0	3	=(2), D
(4) *C,B	0 0 0 3	1	SAME 1
(5) +D,(4)	0 0 0 3	4	+D,(1)
(6) =(5),A	0 0 0 3	5	=(5),A
(7) *C,B	6 0 0 3	1	SAME 1
(8) +D,(7)	6 0 0 3	4	SAME 5
(9) =(8), C	6 0 0 3	5	= (5),C
	6 0 9 3	-	

Однако необходимо помнить, что не все арифметические выражения можно оптимизировать, например, выражение B*C+C*B.

Здесь C*B – избыточная тройка, но мы ее не найдем.

Следовательно, перед проведением оптимизации необходимо выполнить предварительные шаги, а именно: переупорядочивание в лексикографической последовательности. Например, A=1+B+C+2 представить как A=B+C+1+2.

После переупорядочивания можно провести оптимизацию.

Необходимо по возможности переносить одноместные операции к операндам.

Пример:

Имеем выражения:
C=A-B
D=B-A
Запишем тройки

(1)+A,B`	(3)+B, A`
(2)=(1), C	(4)=(3),D

Заменим

(1)+A, B`	(3)+A,B`
(2)=(1), C	(4)=(3)`,D

Здесь «`» – одноместный минус.
Тройки (1) и (3) – одинаковы.

7.2.3. Оптимизация циклов

Оптимизация циклов – одна из самых сложных задач оптимизации.

Сложности:

– отыскание операторов, инвариантных относительно цикла;

– построение эквивалентных выражений;

– размещение эквивалентных выражений в теле цикла и вне его.

Оптимизация циклов выполняется над исходным кодом, что, однако, не всегда реализуется, или близким к нему представлением – в нашем случае это та или иная форма ВФП –

и обладает широким диапазоном возможностей. Например, в следующем коде выражение **a*b** вычисляется в каждой итерации цикла:

```
int v[10];
void f(void)
{
  int i,x,y,z;
  for (i = 0; i < 10; i++)
    v[i] = a * b;
}
```

Этот код можно оптимизировать следующим образом:

```
int v[10];
void f(void)
{
  int i,x,y,z,t1;
  t1 = a * b;
  for (i = 0; i < 10; i++)
    v[i] = t1;
}
```

Другими оптимизациями по отношению к циклам являются:

- замена остаточной рекурсии (tail recursion) итерацией;
- удаление ненужных проверок границ массивов;
- развертка цикла (замена цикла фрагментом последовательного кода).

Вот конкретные примеры более сложной оптимизации цикла.

Назовем операцию *инвариантной* в цикле, если ни один из операндов, от которых она зависит, не изменяется во время работы этого цикла. Одна из важных оптимизаций состоит в том, чтобы вынести инвариантную операцию за границы цикла. Например, если вынести одно умножение из цикла, выполняемого 1000 раз, то на этапе выполнения готовой программы мы сэкономим 999 умножений!

Второй тип оптимизации цикла, который мы описываем здесь, называется *заменой сложных операций* (что называется также «уменьшением мощности операции») и состоит в замене операции на более быструю. Нас главным образом будет интересовать замена умножения типа I*K на сложение, где I – переменная цикла. Для иллюстрации замены сложной операции рассмотрим цикл

FOR I : =A STEP B UNTIL C DO BEGIN . . . TI := I * К - - . END,

где переменная К инвариантна в цикле. Начальное значение I равно значению А. Внутри цикла I *определяется рекурсивно,* то есть только посредством самой себя. (Единственным присваиванием I внутри цикла является ее продвижение на шаг I := 1+В.) Каждый раз, когда I изменяется на В, значение I*К (и Т1) изменяется на В*К. То есть

(I+B)*K = I*K+B*K.

Следовательно, если Т1 нигде больше в цикле не изменяется, мы можем заменить сложную операцию I*К, следующим образом изменяя задание цикла:

Перед циклом вставляются операции Т1 := А*К; Т2 : = В*К, где Т2 – новая временная переменная. Это начальная установка Т1 и вычисление приращения В*К.

Из цикла исключается Т1 : = I*К.

В конце цикла вставляется Т1 := Т1+Т2.

В результате мы получим:

Т1:=А*К; Т2:=В*К;

FOR I : = A STEP B UNTIL C DO

BEGIN ... . . . Tl := T1+T2; END

Итак, внутри цикла мы заменили умножение сложением. Мы предполагаем, конечно, что А и В инвариантны в цикле и I нигде не изменяется в цикле, за исключением места, где она получает приращение. (В действительности в нашей реализации переупорядочивание операций будет несколько иным, так что инвариантность А будет несущественна.)

Некоторые предостережения. Мы предполагали, что В и К имеют тип INTEGER. Никогда не следует заменять умножение, если К имеет тип REAL, так как сложение с плавающей запятой неточно из-за погрешности округления. Эта погрешность имеет тенденцию накапливаться во время многократного выполнения цикла. Вообще I*К намного точнее, чем К+К+ • • • +К (I раз).

Замена умножения может быть полезна, если даже умножение работает так же быстро, как сложение, так как позволяет нам легко находить и выбрасывать несколько раз встречающиеся одинаковые операции. Этим самым в пределах

цикла мы в некоторой ограниченной форме проводим исключение лишних операций. Это особенно полезно при оптимизации вычисления переменных с индексами.

Следует также помнить, что замена сложных операций не всегда приводит к оптимизации. На самом деле это может даже привести к замедлению! Ниже соответственно слева и справа приводится программа до и после замены:

```
FOR I: = 1 STEP 1 UNTIL 10 DO
CASE I OF
        K : =1 * 1;
        K :=I * 2;
        ……
       K :=I * 10;
ENDCASE;
```

```
Tl :=1; T2 :=2; . . . T10 : = 10;
FOR I: = 1 STEP 1 UNTIL 10 DO
BEGIN CASE I OF
        K :=T1;
        K : = T2,
        ……
       K : = T10;
ENDCASE;
T1 : = T1 + 1; ... T10 :=T10+10;
END;
```

В программе слева на каждом шаге цикла выполняется одно умножение. Теперь в программе справа внутри цикла нет умножений, но зато выполняются 10 сложений!

Замена сложных операций почти всегда приводит к получению более эффективной программы. Это не так только для циклов, состоящих из нескольких частей, из которых лишь немногие выполняются на каждом шаге цикла.

Возможны также и иные приемы оптимизации, например «Удаление недостижимого кода». Недостижимый код – это некоторая последовательность инструкций программы, которая недостижима ни по одному пути в программе. Он может образоваться как следствие предыдущих операций оптимизации, кода условной отладки или частых изменений программы многими программистами.

7.3. Контрольные вопросы по теме «Машинно-независимая оптимизация»

1. В чем состоит основная задача МНО?
2. Каковы основные направления МНО?

3. Почему после оптимизации уменьшается объем генерируемого кода ассемблера?
4. Что такое «коэффициент зависимости» и в каком виде оптимизации он присутствует?
5. Почему важна оптимизация циклов?
6. Что такое – уменьшение «мощности» операций?
7. Какие проблемы могут возникнуть при оптимизации арифметических выражений?

8. ГЕНЕРАЦИЯ КОДОВ

8.1. Формы объектного кода

Возможны две основные формы объектного кода:

- абсолютные (машинные) команды, помещенные в фиксированные ячейки (после окончания компиляции такая программа немедленно выполняется);
- программа на ассемблере (автокоде), ее придется потом транслировать системным ассемблером.

Первый вариант наиболее экономичен в отношении расходуемого времени. Компиляторы, генерирующие абсолютный код, целесообразно применять там, где через машину проходит масса мелких программ, при отладке которых многократно используется компилятор. В настоящее время такие компиляторы редки.

Проще всего получить объектную программу на ассемблере. Можно также генерировать макровызовы, а соответствующие макроопределения предварительно написать на ассемблере. Это позволяет также уменьшить объем компилятора.

8.2. Система адресации, используемая при описании генерации кода

Чтобы показать, как генерируются ассемблерные команды, воспользуемся гипотетической машиной с одним сумматором, в котором выполняются все арифметические действия, и семью индексными регистрами. Команды, приведенные ниже, типичны для машин средней мощности. Имеются команды: загрузить сумматор или регистр из памяти, записать в память содержимое сумматора или регистра, прибавить (вычесть и т.д.) число к сумматору. Числа в машине можно представить как в формате с фиксированной точкой, так и в формате с плавающей точкой. Конкретные сведения о форматах нас интересовать не будут. Число в любом из индексных регистров должно быть в формате с фиксированной точкой.

8.3. Генерация команд для простых арифметических выражений

Программы генерации команд пользуются описанием в дескрипторе (в таблице символов или другой таблице) каждой переменной или временного значения. В описании указан тип переменной, ее адрес во время выполнения программы и другая необходимая информация. Покажем в общих чертах идеи генерации команд и как и что меняется в зависимости от формата внутренней исходной программы.

Команды	***Смысл***
LREG i, M	Занести содержимое ячейки M в регистр
STORE M	Занести содержимое сумматора в ячейку M
SREG i, M	Занести содержимое регистра i в ячейку M
LACCR i	Занести содержимое регистра i в сумматор
LRACC i	Занести содержимое сумматора в регистр i
ADD (ADDF) M	Прибавить содержимое ячейки M к сумматору
SUB (SUBF) M	Вычесть содержимое ячейки M из сумматора
MULT (MULTF) M	Умножить содержимое сумматора на содержимое ячейки M
DIV (DIVF) M	Разделить содержимое сумматора на содержимое ячейки M. При делении дробная часть результата отбрасывается
FIX	Преобразовать содержимое сумматора из формы с плавающей точкой в форму с фиксированной точкой
FLOAT	Преобразовать содержимое сумматора из формы с фиксированной точкой в форму с плавающей точкой
ABS	Сделать знак содержимого сумматора +
CHS	Изменить знак содержимого сумматора
B M	Перейти к команде в ячейке M
BN M	Перейти к команде в ячейке M, если сумматор отрицательный (смысл других команд перехода BP, BZ и т.д. очевиден)

Рис. 8.1. *Символические команды гипотетической машины*

ADD, SUB, MULT, DIV – команды для чисел с фиксированной точкой.
ADDF, SUBF, MULTF, DIVF – команды для чисел с плавающей точкой.

Для простоты мы в этом разделе будем предполагать, что каждая переменная представлена своим символическим именем (а не указателем на описание) и что генерируются символические команды на автокоде. Для генерации команды на ассемблере с операцией X и операндом Y будет вызываться процедура GEN (X, Y), где X и Y – переменные типа STRING (например, GEN ('ADD', 'GAB')).

Рассматриваются только такие арифметические выражения, в которых используются операторы +, —, / и унарный минус, а все операнды являются простыми переменными целого типа. Мы покажем, как генерируются команды для тетрад, триад, деревьев, используя в качестве примера выражение A*((A*B+C) – C*D).

8.3.1. Генерация кода для тетрад

Тетрады расположены в том порядке, в котором должны выполняться операции. Поэтому мы просматриваем их последовательно одну за другой и для каждой генерируем команды. Чтобы устранить ненужные команды LOAD и STORE, нам придется постоянно следить за содержимым сумматора, в котором при выполнении программы производятся все арифметические действия. Для этого введем глобальную переменную ACC типа STRING. Значение переменной ACC ***во время компиляции*** соответствует состоянию сумматора ***во время выполнения программы*** именно в тот момент, когда выполнится последняя сгенерированная команда. Если ACC содержит строку ' ' (пусто), сумматор свободен; в противном случае в ACC содержится имя переменной или временного значения, находящегося в сумматоре (после того, как сгенерированная команда будет выполнена при работе программы).

Чтобы сгенерировать команды (если это необходимо), которые загружают во время выполнения программы переменную X или Y в сумматор, должна быть вызвана программа

GETINACC(X, Y). Например, программа GETINACC будет вызвана при генерации команд для вычисления X*Y, причем один из двух операндов уже может быть в сумматоре. Некоторые операторы (такие, как X/Y) не являются коммутативными в машине и требуют, чтобы в сумматоре был первый операнд. Поэтому условимся, что в результате вызова GETINACC (X, ' ') должна генерироваться команда занесения X в сумматор. Наконец, заметим, что программе GETINACC, возможно, придется сформировать команды, которые сохранят содержимое сумматора, если в данный момент сумматор не пуст.

Как упоминалось ранее, тетрады просматриваются последовательно и генерируются команды для каждой из них. Предположим, что счетчик i следит за последовательным перебором тетрад и что на четыре поля i-й тетрады можно сослаться с помощью QD(i).OP, QD(i).OPER1, QD(i).OPER2 и QD(i).RESULT.

```
PROCEDURE GETINACC (X, Y);  Генерирует команды для
                            занесения X или Y в
                            сумматор (X, если Y=' ').
STRING X, Y;
BEGIN STRING T;             T — временная переменная.
IF ACC =' ' THEN            Если сумматор пуст,
                            генерируется команда LOAD
                            X и соответствующим
                            образом изменяется
                            глобальная переменная ACC
BEGIN GEN ('LOAD', X);
ACC: = X; RETURN
END;
IF ACC = Y THEN             Если Y уже в сумматоре,
                            операнды меняются местами.
BEGIN T: = X; X:=Y;
Y: =T END
ELSE IF ACC ≠ X THEN        Если X не в сумматоре, то
                            генерируются команды для
                            запоминания сумматора и
                            загрузки X.
BEGIN GEN ('STORE', ACC);
GEN ('LOAD', X);
ACC: =  X END; END
```

Ниже приводятся генераторы команд для каждого из операторов +, -, *, /.

В генераторе команд для коммутативного оператора "+" обратите внимание на тот факт, что, если второй операнд уже находится в сумматоре, программа GETINACC поменяет местами два операнда в i-й тетраде. В последнем генераторе для унарного минуса имеется только один операнд; следовательно, вторым аргументом при вызове GETINACC будет значение пусто ''.

```
Генератор команд для оператора
"+"
GETINACC (QD(i).OPER1,QD(i).OPER2); Генерация команды
                                    занесения операнда в
                                    сумматор.
GEN ('ADD',QD(i).OPER2);            Генерация   команды
                                    ADD.
ACC := QD(i).RESULT                 Установка состояния
                                    сумматора.
Генератор команд для оператора
"*"

GETINACC                            Занесение   операнда
(QD(i).OPER1,QD(i).OPER2);          в сумматор
GEN ('MULT',QD(i).OPER2);           Генерация команды
                                    умножения.
ACC := QD(i).RESULT                 Установка состояния
                                    сумматора.
Генератор команд для оператора
"-"
GETINACC(QD(i).OPER1, ' ');         Первый операнд
                                    команды SUB должен
                                    быть в сумматоре.
GEN ('SUB',QD(i).OPER2);            Генерация   команды
                                    SUB.
ACC := QD(i).RESULT                 Установка состояния
                                    сумматора.
Генератор команд для оператора
"/"
GETINACC (QD(i).OPER1, ' ');        Первый операнд
                                    должен быть в
                                    сумматоре.
GEN ('DIV',QD(i).OPER2);            Генерация команды
                                    деления.
ACC := QD(i).RESULT                 Установка  состояния
                                    сумматора.
```

В колонке 1 на рис. 8.2 изображены тетрады для приведенного выше арифметического выражения; в колонке 2 – соответствующие команды, сгенерированные для каждой тетрады; в колонке 3 показано значение глобальной

переменной ACC непосредственно после генерации команд. Заметим, что хотя во время компиляции имеется описание каждой временной переменной Ti, не следует выделять ячейки памяти для тех временных переменных, которые остаются в сумматоре во время их существования.

Тетрады	*Сгенерированные команды*		*ACC*
* A, B, T1	LOAD	A	
	MULT	B	T1
+ T1, C, T2	ADD	C	T2
* C, D, T3	STORE	T2	
	LOAD	C	
	MULT	D	T3
- T2, T3, T4	STORE	T3	
	LOAD	T2	
	SUB	T3	T4
* A, T4, T5	MULT	T4	T5

Рис. 8.2. *Генерация кода для тетрад*

8.3.2. Генерация кода для триад

Тетрады неудобны в том отношении, что описание каждой временной переменной хранится на протяжении всей компиляции. При работе с триадами в этом нет необходимости. Кроме того, внутренняя исходная программа получается более компактной, поскольку в триаде только три поля. При использовании триад нам все-таки требуется описание каждой временной переменной, но такое описание необходимо хранить лишь до тех пор, пока генерируются команды, которые на него ссылаются. Например, когда генерируются команды для триады

(10) * A, B,

генерируется также и описание результата. Затем после генерации команд для последней триады, которая ссылается на триаду (10), мы уничтожаем это описание.

Если зоны существования временных переменных попарно не пересекаются или вложены друг в друга, для хранения их описаний во время компиляции можно использовать стек; в противном случае потребуется более сложная схема выделения памяти для описаний и освобождения ее. Мы приведем пример генерации команд для первого случая. Кроме того, будем считать, что на каждое временное значение есть только одна ссылка.

Во время генерации команд мы храним в стеке TRIP номера триад, для которых команды уже сгенерированы, но полученные результаты еще не использовались. Параллельный стек TEMP содержит соответствующие имена Тк, присвоенные результатам. Счетчик j содержит номер текущего элемента в TRIP и TEMP.

Перед генерацией команд для триады i следует проверить оба операнда, и если какой-то из них является ссылкой на предыдущую триаду, его нужно заменить соответствующим именем временной переменной, присвоенным этой триаде. Это выполняет процедура, приведенная ниже. Параметры этой процедуры: X – поле операнда, который следует проверить; Y будет при возврате содержать имя переменной или временной переменной.

После того как сгенерированы команды для триады i, следует образовать имя для описания результата. Имя результата и номер триады заносятся в «параллельные» стеки TEMP и TRIP. Поскольку результат выполнения только что сгенери-рованных команд также находится в сумматоре, следует также изменить ACC, описывающую состояние сумматора. Эту работу выполняет процедура NEWTEMP.

При генерации команд последовательно просматриваются все триады с использованием счетчика i. Мы предполагаем, что ссылки на i-ю триаду имеют вид TR(i).OP, TR(i).OPERl и TR(i).OPER2. Генераторы команд для различных операторов похожи на те, которые использовались для тетрад. Различие состоит в том, что все операции и в том числе поиск имени, соответствующего номеру триады, исключение временных переменных из стека, образование нового имени для результата и занесение его в стек, станут более трудоемкими.

```
PROCEDURE FIXTEMP (X, Y);
BEGIN
IF X ссылается на триаду k THEN  Если X есть ссылка
                                 на какую-либо триаду
                                 k, то k ищется в
                                 стеке TRIP и соот-
                                 ветствующее имя из
                                 TEMP заносится в Y.
BEGIN FOR m:=j STEP — 1 UNTIL 1  Начиная с текущего j
                                 спускаемся вниз по
                                 стеку TRIP.
DO
IF TRIP(m) = k THEN
GOTO F;
F: Y:=TEMP (m)
END
ELSE Y: = X;                     В противном случае X —
                                 имя переменной, и
                                 оно заносится в Y.
END
PROCEDURE NEWTEMP;
BEGIN
T: = имя новой временной         Образовать имя новой
переменной;                      временной
                                 переменной.
j:=j+1;
TEMP(j):=T; TRIP(j): =i;         Занести имя и номер
                                 триады i в стек(i —
                                 глобальная
                                 переменная).
ACC:=T;                          Фиксация состояния
                                 сумматора.
END
```

Так как мы предполагаем, что зоны существования двух временных переменных не пересекаются или вложены друг в друга, то имя той временной переменной, на которую могут сослаться генерируемые в данный момент команды, обязательно находится в одном из двух верхних элементов стека. Поэтому его можно найти, проверив только два верхних элемента стека и не просматривая весь стек. Это упрощает процедуру FIXTEMP.

Генератор команд для триады + (генератор для триады * аналогичен).

```
FIXTEMP (TR (i).OPER1, T1); Занести имена операндов в
                            T1 и T2.
FlXTEMP(TR(i).OPER2, T2);
GETINACC(T1, T2);
GEN ('ADD', T2);            Генерация команд.
IF  TR  (i).OPER1  ссылаетсяЕсли операнд ссылается на
на триаду                   триаду,
THEN j:=j—1 ;               стираем его имя в стеке;
IF  TR  (i).OPER2  ссылаетсяоно больше не понадобится.
на триаду
THEN j: =j—1
NEWTEMP                     Фиксация нового временного
                            имени и состояния
                            сумматора.
```

На рис. 8.3 показана генерация команд для триад соответствующих выражению A*((A*B+C) – C*D). Колонка 1 содержит триады; в колонке 2 показано, что получилось после замены операндов, которые ссылаются на триады, соответствующие именам временных переменных; в колонке 4 приводятся стеки TRIP и TEMP после обработки триады; в колонке 5 показано состояние сумматора после обработки триады. Заметим, что команды полностью совпадают с теми, которые были сгенерированы для тетрад.

	Триада	**После замены**	**Команды**		**Стек**	**ACC**
(1)	* A, B	* A, B	LOAD	A		
			MULT	B	1, T1	T1
(2)	+ (1), C	+ T1, C	ADD	C	2, T2	T2
(3)	* C, D	*C, D	STORE	T2		
			LOAD	C	3, T3	
			MULT	D	2, T2	T3
(4)	- (2),(3)	- T2, T3	STORE	T3		
			LOAD	T2		
			SUB	T3	4, T4	T4
(5)	* A, (4)	* A, T4	MULT	A	5, T5	T5

Рис. 8.3. *Генерация кода для триад*

Несколько замечаний относительно образования и исключения описаний (имен) временных переменных. Если не предполагается никакой другой обработки триад после того, как

сгенерированы команды, то триаду можно заменять описанием ее результата. Это значит, что, как только сгенерированы команды для триады i, можно занести в одно или в несколько полей TR(i) имя, присвоенное результату триады i. Тогда не нужен будет стек и не потребуется уничтожать описания.

Если на временную переменную может быть более одной ссылки или если зоны существования временных переменных пересекаются, то мы должны знать, где кончается каждая зона. Это необходимо не только для исключения описания временной величины, а еще и потому, что, пока ее значение продолжает использоваться в программе, мы должны генерировать команды для его запоминания.

8.3.3. Генерация команд для дерева

Последовательность из N триад для одного арифметического выражения можно рассматривать как дерево с корневым кустом TR(n). На рис. 5.5 были изображены триады и соответствующее дерево для выражения. Будем теперь генерировать команды начиная с корневого узла дерева, то есть с последней, а не с первой триады. Тогда у нас будет большая свобода в выборе последовательности, в которой будут генерироваться команды, и программа получится более эффективной, чем раньше.

Мы реализуем рекурсивную процедуру COMP(i), которая предназначена для компиляции команд для поддерева с корнем TR(i). Выбор тех или иных действий в процедуре COMP(i) зависит от оператора TR(i).OP и его операндов TR(i).OPER1 и TR(i).OPER2. Ниже в таблицах-матрицах имеется сводка действий, которые должны быть выполнены для различных операторов. Рассмотрим таблицу для оператора +. Если оба операнда – TR(i).OPER1 и TR(i).OPER2 – являются именами переменных, то генерируется команда загрузить OPER1 в сумматор и команда прибавить OPER2 к сумматору.

Если OPER1 – переменная, а OPER2 – поддерево, корнем которого является оператор, то TR(i).OPER2 будет индексом того корня в списке триад TR. Чтобы сгенерировать команды для этого поддерева, рекурсивно вызывается процедура

COMP. Будут сгенерированы команды, которые вычислят и оставят в сумматоре значение выражения. После возврата из COMP генерируется команда ADD с операндом OPER1.

Если оба операнда – поддеревья, то вызывается COMP, чтобы скомпилировать команды для первого поддерева. Затем в поле TR(i).OPER1 заносится указание о том, что значение первого операнда находится в сумматоре. REPEAT означает, что для определения дальнейших действий следует снова воспользоваться той же матрицей. Теперь выполняются действия, указанные для случая, когда TR(i).OPER1 = ACC и TR(i).OPER2 = поддерево. То есть генерируется новое временное имя, генерируется команда STORE для запоминания содержимого сумматора, компилируются команды для второго операнда и, наконец, генерируется команда ADD.

В приведенной ниже таблице показана последовательность действий и команды, сгенерированные в результате вызова COMP(5) для дерева, которое изображено на рис. 5.5. В колонке 1 перечислены обрабатываемые узлы, а в колонке 2 – их текущее состояние. В колонках 3, 4 и 5 приводятся действия, выполняемые перед генерацией команд, сгенерированные команды и действие, выполняемое после генерации. Так как в триаде (4) – (2), (3) представилась возможность вначале сгенерировать команды для второго операнда, нам удалось сэкономить операции LOAD и STORE.

Триада	**Действие**	**Команда**	**Действие**
(5)* A, (4)	COMP(4)		
(4) - (2), (3)	COMP(3)		
(3)* C, D		LOAD C	
		MULT D	RETURN
(4) - (2), ACC		STORE T1	COMP(2)
(2) + (1), C	COMP(1)	LOAD A	
(1) * A, B		MULT B	RETURN
(2) + ACC, C		ADD C	RETURN
(2) - ACC, T1		SUB T1	RETURN
(5)* ACC, A		MULT A	RETURN

Ниже приводится пример матрицы генерации кодов для оператора "+":

OPER2 OPER1	ACC	Переменная	Целое число(поддерево)
ACC	невозможно	GEN ('ADD', TR (i).OPER2)	T: = нов. врем, имя; GEN('STORE',T) COMP(TR(i).OPER2) GEN ('ADD', T)
переменная	GEN('ADD' TR(i).OPERl)	GEN('LOAD' TR(i).OPERl) GEN('ADD' TR(i).OPER2)	COMP(TR(i).OPER2) GEN('ADD',TR(i).OPERl)
целое число (поддерево)	невозможно	COMP(TR(i).OPERl) GEN('ADD' TR(i).OPER2)	COMP(TR(i).OPERl) TR(i).OPERl := ACC REPEAT

Если семантические программы могут генерировать тетрады или триады, из которых в том же порядке генерируются команды, то они могут с таким же успехом непосредственно генерировать команды. То есть вместо схемы

исходная программа—> тетрады—> команды

имеем

исходная программа—> команды.

Конечно, семантические программы становятся значительно сложнее, но компилятор в целом получается более быстрым.

8.3.4. Получение более оптимального объектного кода

Одна из трудностей, с которой непременно сталкиваются при генерации кода, состоит в том, что существует немало искусных приемов, и их, казалось бы, надо обязательно использовать, чтобы сэкономить одну-другую команду в объектной программе. Эти приемы зависят от языка той конкретной машины, для которой генерируется программа, и поэтому они почти не поддаются стандартизации. Это означает, что при их реализации в каждом фрагменте программы,

который генерирует команды для того или иного оператора, нужно выделять особые случаи и для каждого из них различным образом генерировать коды. В результате система генерации объектного кода становится чрезвычайно громоздкой. Достаточно привести несколько примеров.

В составе машинных команд может быть команда MOVE, которая пересылает содержимое одной ячейки памяти в другую ячейку. Тогда, если нужно сгенерировать команды для тетрады (:= A, ,B) и A не находится в сумматоре, следует сгенерировать команду

MOVE A, B вместо LOAD A
STORE B.

Однако заметим, что, если следующей тетрадой будет, например, тетрада (+ A, C, T1), следует предпочесть команды LOAD и STORE, так как значение переменной A в конечном итоге должно оказаться в сумматоре.

В некоторых машинах имеются команды «Сложение непосредственное» и «Загрузка непосредственная», где величина, которую надо прибавить или загрузить, является адресной частью самой команды. Тогда для тетрады (+ A, 3, T1) следует сгенерировать команды

LOAD A вместо LOAD A
ADDI 3 ADD L3,

где L3 – ячейка, содержащая целое число 3. Кроме экономии ячейки, каждый раз при выполнении этих команд экономится обращение к памяти.

В машине может быть обратное деление INDIV (или вычитание). И в таком случае, когда делитель (или вычитаемое) находится в сумматоре, можно сэкономить команды LOAD и STORE, генерируя одну команду

INDIV A вместо LOAD A
DIV B

для тетрады (/ A, B, T1), поскольку B уже находится в сумматоре.

Можно извлечь выгоду из команды LOADN (Загрузка отрицательная). Тогда для тетрады (- A, ,T1), если A не находится в сумматоре, генерируется команда

LOADN A вместо LOAD A
CHS.

Некоторые функции, такие, как ABS и ENTIER в Алголе, зачастую лучше встроить в программу и не генерировать вызов подпрограммы.

8.4. Память для данных элементарных типов

Прежде чем перейти к конкретным действиям по генерации кодов ассемблера, мы должны представить себе, каким образом те операнды, которые мы видим, например, в тетраде (1)+ A, B, T1, будут представлены во время компиляции. Ясное дело, что никаких имен типа A, B, T1 и т.д. там быть не может – все идентификаторы находятся в соответствующих элементах таблицы идентификаторов. И нам необходимо иметь представление о форме существования операндов во время компиляции.

Естественно, что типы данных исходной программы должны быть отображены на типы данных машины. Для некоторых типов это будет соответствием один к одному (целые, вещественные и т.д.), для других могут понадобиться несколько машинных слов. Коротко мы отметим следующее:

1. Целые переменные обычно содержатся в одном слове или ячейке области данных; значение хранится в виде стандартного внутреннего изображения целого числа в машине.

2. Вещественные переменные обычно содержатся в одном слове. Если желательна большая точность, чем возможно представить в одном слове, то может быть употреблен машинный формат двойного слова с плавающей запятой.

3. Булевы или логические переменные могут содержаться в одном слове, причем нуль изображает значение FALSE, а не нуль или 1 – значение TRUE. Конкретное представление для TRUE и FALSE определяется командами машины, осуществляющими логические операции. Для каждой булевой переменной можно также использовать один бит и разместить в одном слове несколько булевых переменных или констант.

4. Указатель – это адрес другого значения (или ссылка на него). В некоторых случаях бывает необходимо представлять указатель в двух последовательных ячейках; одна ячейка

содержит ссылку на описатель или является описателем текущей величины, на которую ссылается указатель, в то время как другая указывает собственно на значение величины. Это бывает необходимо в случае, когда во время компиляции невозможно определить для каждого использования указателя тип величины, на которую он ссылается.

Отдельный разговор – о формах представления областей памяти для хранения массивов. Вычисление адресов элементов самых разнообразных видов массивов – это одна из самых сложных задач при работе с ними.

8.5. Векторы и матрицы

Элементы векторов (одномерных массивов) обычно помещаются в последовательных ячейках области данных в порядке возрастания или убывания адресов. Порядок зависит от машины и ее системы команд. Будем предполагать, что каждый элемент массива (вектора) занимает в памяти одну ячейку.

Мы используем более общепринятый стандартный возрастающий порядок размещения элементов, то есть элементы массива, определенного описанием ARRAY A [1:10], размещаются в порядке A [1], A [2], ..., A [10].

Матрицы (двумерные массивы)

Существует несколько способов размещения двумерных массивов. Обычный способ состоит в хранении их в области данных по строкам в порядке возрастания, то есть (для массива, описанного как

ARRAY A[1:M, 1:N]) в таком порядке:

A[1,1],A[1,2],..,A[1,N], A[2,1],A[2,2],..,A[2,N],.., A[M,1],..,A[M,N]

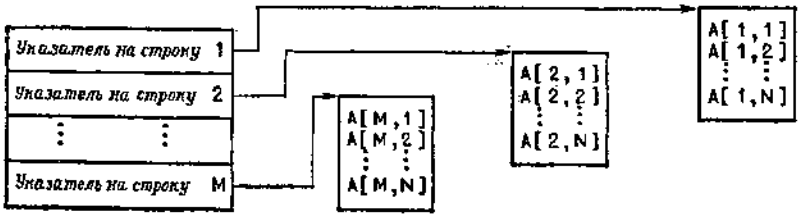
Вид последовательности показывает, что элемент A[i, j] находится в ячейке с адресом:

ADDRESS (A[1,1])+(i-1)*N+(j-1), который мы запишем в виде:

(ADDRESS (A[1,1]) -N-1) + (i*N+j).

Первое слагаемое является константой (CONSTPART), и его требуется вычислить только один раз. Таким образом, для определения адреса A[i,j] необходимо выполнить одно умножение и два сложения. Язык Фортран IV требует размещения массивов по столбцам, а Паскаль хранит массив по строкам.

Второй метод состоит в том, что выделяется отдельная область данных для каждой строки массива и имеется вектор указателей для этих областей данных. Элементы каждой строки размещаются в соседних ячейках в порядке возрастания. Так, описание ARRAY A[1:M,1:N] порождает вот такое представление – указатель – область хранения.



Вектор указателей строк хранится в той области данных, с которой ассоциируется массив, в то время как собственно массив хранится в отдельной области данных. Адрес элемента массива A[i,j] есть CONTENTS (адрес вектора+1-1)+(j-1).

Первое преимущество этого метода состоит в том, что при вычислении адреса не нужно выполнять операцию умножения. Другим является то, что не все строки могут находиться в оперативной памяти одновременно. Указатель строки может содержать некоторое значение, которое вызовет аппаратное или программное прерывание в случае, если строка в памяти отсутствует. Когда возникает прерывание, строка вводится в оперативную память на место другой строки.

Когда известно, что матрицы разреженные (большинство элементов нули), используются другие методы, например хеширование. В хеш-таблице хранятся только ненулевые элементы матрицы.

Адресацию элементов многомерных массивов мы опускаем. Приведем лишь краткое описание тех терминов, которые иногда могут встречаться в других разделах:

- BASELOC – адрес в памяти для первого элемента многомерного массива.
- Значение CONSPART (постоянная составляющая) необходимо вычислить только один раз и запомнить, так как оно зависит лишь от нижних и верхних границ.
- Для многомерных массивов используется также и значение VARPART (переменная составляющая), которая зависит от значений индексов i,j, ...m и от размеров измерений d1,d2, • • •,dn.

Информационный вектор для массива

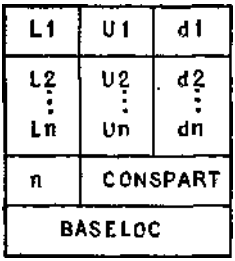


Рис. 8.4. *Информационный вектор для массива* $A[L_1:U_n],...,[L_n:U_n]$

В Фортране верхняя и нижняя границы массивов известны во время трансляции.

Поэтому компилятор может выделить память для массивов и сформировать команды, ссылающиеся на элементы массива, используя верхние и нижние границы и постоянные значения $d_1,d_2,\ldots,d_n$. В Алголе и PL/1 это невозможно, так как границы могут вычисляться во время счета.

Таким образом, нам нужен описатель для массива, содержащий необходимую информацию. Этот описатель для массива называется dope vector, или информационный вектор. Информационный вектор имеет фиксированный размер, который также известен при компиляции, следовательно, память для него может быть отведена во время компиляции в области данных, с которой

ассоциируется массив. Память для самого массива не может быть отведена до тех пор, пока во время счета не выполнится вход в блок, в котором описан массив. При входе в блок вычисляются границы массива и производится обращение к программе распределения памяти для массивов. Эта программа вычисляет число необходимых ячеек, вызывает GETAREA, чтобы выделить область данных нужного объема, и вносит в информационный вектор необходимую информацию. На рис. 8.5 представлена схема работы программы распределения памяти для массива.

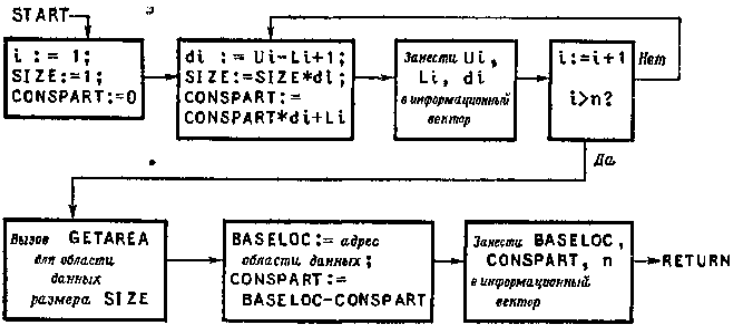


Рис. 8.5. *Программа распределения памяти для массивов*

Какая информация заносится в информационный вектор? Для предложенной выше n-мерной схемы нам как минимум нужны знания $d_2,\ldots,d_n$ и CONSPART. Если перед обращением к массиву нужно проверять правильность задания индексов, то следует также занести в информационный вектор значения верхних и нижних границ.

Параметры программы: n (число индексов), нижние и верхние границы $L_1U_1, \ldots, L_nU_n$ и адрес информационного вектора.

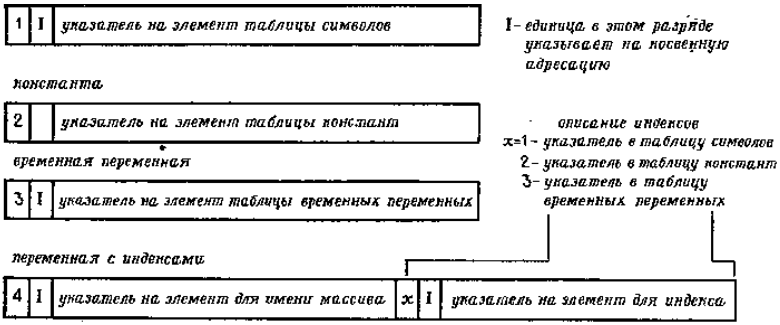
В данном конспекте мы не будем останавливаться на других проблемах с выделением памяти под разного вида операнды, такие, как строки, структуры (записи) и т.д., так как вариантов языков и компиляторов очень много. В каждом отдельном случае при возникновении проблемы надо обращаться к конкретным источникам по конкретному языку.

8.6. Формы представления операндов в ВФП

Все формы внутреннего представления (ВФП) обычно содержат в себе два программных объекта – операторы и операнды, представленные в той или иной форме – например структурах. В дальнейшем мы будем использовать такие операторы, как +, -, /, BR (Branch – переход) и т.д. Внутри компилятора они, конечно, представлены целыми числами. Таким образом, мы могли бы использовать число 4 вместо +, 5 вместо -, 6 вместо / и т.д.

Операндами, с которыми мы имеем дело, являются лишь простые идентификаторы (переменных, процедур и т.д.), константы, временные переменные, генерируемые самим компилятором, и переменные с индексами. Если все идентификаторы и константы хранить в одной общей таблице, то, за исключением переменных с индексами, каждый операнд может представляться указателем на соответствующий элемент в таблице символов. Если описания операндов находятся в нескольких таблицах, то в первой ячейке необходимо иметь различные целые числа для разных типов операндов.

В поле операнда можно предусмотреть признак косвенной адресации и не заводить для этой цели отдельный оператор. То есть сама форма операнда может указывать, что данное значение есть адрес того значения, которое на самом деле требуется.



Итак, операнд представляет собой структуру, в которой обязательно имеется указатель на тот или иной символ программы, который расположен в таблице идентификаторов, содержащей аргументы и дескрипторы.

В общем поле дескриптора обязательно имеются те или иные поля, содержащие информацию о текущем представлении областей памяти, в которых эти величины располагаются во время компиляции.

8.7. Области данных и дисплеи

Областью данных (ОД) является ряд последовательных ячеек – блок оперативной памяти, – выделенный для данных, каким-то образом объединенных логически. Часто (но не всегда) все ячейки области данных принадлежат одной и той же *области действия* в программе на исходном языке; к ним может обращаться один и тот же набор инструкций (то есть этой областью действия может быть блок или тело процедуры). Во время компиляции ячейка для любой переменной времени счета может быть представлена упорядоченной парой чисел (номер области данных, смещение), где номер области данных – это некоторый единственный номер, присвоенный области данных, а смещение – это адрес переменной относительно начала области данных. Так, первая ячейка области данных №3 представляется парой (3,0), вторая – (3,1) и т.д.

Область памяти

(3,0)	Первая ячейка в области 3
(3,1)	Вторая ячейка в области 3
.....	.....

Рис. 8.6. *Область данных ОД (Data Area)*

Когда мы генерируем команды обращения к переменной, эта упорядоченная пара переводится в действительный адрес переменной. Это обычно выполняется установкой *адреса базы* (машинного адреса первой ячейки) области данных в так называемом базовом регистре и обращению к переменной

по адресу, равному смещению плюс содержимое базового регистра. Пара (номер области данных, смещение) таким образом переводится в пару (адрес базы, смещение).

Области данных делятся на два класса – *статический* и *динамический.* Статическая область данных имеет постоянное число ячеек, выделенных ей перед началом счета. Эти ячейки выделяются на все время счета. Следовательно, на переменную в статистической области данных возможна ссылка по ее абсолютному адресу вместо пары (адрес базы, смещение). Ниже мы будем рассматривать более сложный вариант работы с ОД, а именно: динамический вариант. Он будет подразумевать, что память под каждую отдельную ОД мы будем выделять только тогда, когда происходит обращение к конкретной процедуре. Рассмотрим пример некоторого фрагмента программы со вложенными процедурами.

```
BEGIN   Это начало главной программы (MAIN)

PROCEDURE A;
BEGIN
PROCEDURE B;
        BEGIN...END;
PROCEDURE C;
        BEGIN...END;
END;
PROCEDURE D;
BEGIN...END;
.....
END;
```

Рис. 8.7. *Структура программы с вложенными процедурами*

В данном фрагменте в главной программе существует процедура A с вложенными в нее процедурами B и C.

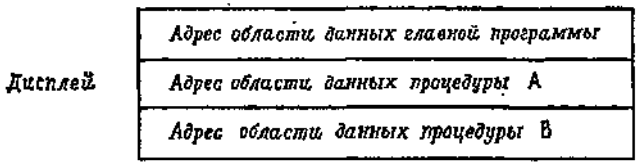
Процедура D вложена только в главную программу.

Динамическая область данных во время счета существует не всегда. Она появляется тогда, когда имеется обращение к данной процедуре, и исчезает после выхода из нее, и всякий раз, когда она исчезает, теряются все величины, хранящиеся

в ней. Примером может служить область данных, содержащая переменные некоторой процедуры в Алголе. Процедура, к которой произошло обращение, вызывает программу GETAREA, чтобы выделить память для своей области данных постоянной длины. Непосредственно перед возвратом в вызывающую программу процедура вызывает программу FREEAREA для освобождения памяти.

Заметим, что, возможно, процедуре выделяются не одни и те же ячейки памяти для ее области данных. Однако GETAREA всегда выдает адрес базы выделенной области данных, и процедура всегда хранит его в одной и той же ячейке, например, BASE_ADDRESS (BA). Таким образом, адресом любой переменной в области данных всегда является CONTENTS(BA)+OFFSET – содержание базового регистра+смещение.

Такая область данных по сути принадлежит не собственно процедуре, а времени исполнения процедуры. В любой момент выполнения процедуры возможны обращения к переменным из нескольких областей данных. Например, если для главной программы и для каждой из процедур существуют различные области данных, то при выполнении процедуры B возможно обращение к областям данных процедур A, B и главной программы. Чтобы иметь возможность ссылаться на них, мы собираем адреса этих областей данных в таблицу, которую назовем *дисплеем*. Для этого частного примера при выполнении процедуры B дисплей будет иметь вид:



Таким образом, когда мы создаем области данных (получаем для них память), мы имеем постоянное место для запоминания их адресов. Где же должен помещаться сам дисплей? Как он будет заполняться?

Вообще при исполнении программы может понадобиться несколько дисплеев. Предположим, что в вышеприведенном примере в процедуре B есть вызов процедуры D. Тогда мы должны создать дисплей, содержащий только адреса областей данных для D и главной программы, используемых в D. Мы должны сохранить нетронутым предыдущий дисплей; он будет нужен снова, когда выполнение D завершится и выполнится возврат в B.

Существует два основных способа размещения самого дисплея:

1. Поскольку дисплей то необходим, то не нужен – его часто хранят в стеке: нужен – помещаем в стек, нет – удаляем.

2. Второй способ для нас представляется нам более понятным. Так как с каждой областью данных всегда связан некоторый дисплей, то очевидно, что дисплей ОД мы можем разместить в самой ОД – в самом ее начале. Естественно, что мы должны будем предусмотреть изменение смещения для переменных области. Таким образом, адрес *активного дисплея* всегда совпадает с адресом активной ОД (то есть в которой выполняются операции чтения и записи).

Введем также понятие некоторой глобальной переменной ACTIVEAREA, которая всегда содержит адрес активного дисплея. Обычно для этой цели выделяют некоторый заранее выбранный регистр (назначение по соглашению).

Если компилятор знает все характеристики переменных во время компиляции и место их расположения, то он может сгенерировать полностью команды обращения к переменным, основываясь на этих характеристиках.

8.8. Адресация операндов на основе ОД и дисплеев

В предыдущем разделе описывалась в общих чертах идея генерации объектного кода для различных форм внутреннего представления исходной программы.

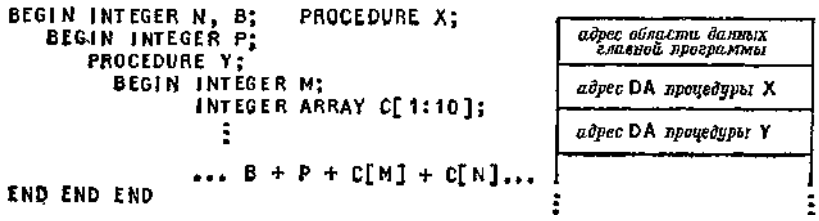


Рис. 8.8. *Сегмент программы и дисплей для процедуры Y*

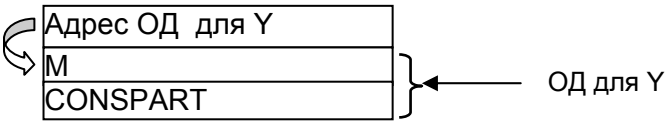


Рис. 8.9. *Область данных дисплея процедуры Y*

Для простоты изложения каждый операнд у нас представляется символическим именем и располагается в одной ячейке памяти. В связи с этим смещение, с помощью которого адресуется переменная, будет измеряться в ячейках (вместо обычно применяемого смещения в байтах).

В примере используется арифметическое выражение с операндами целого типа, чтобы сосредоточить внимание только на этой задаче. Внутреннее представление исходной программы – тетрады.

В любой момент генерации кода мы генерируем команды для тела одной процедуры. Предположим, что в соответствующий момент выполнения готовой программы адрес области данных (DATA AREA – DA) и дисплея этой процедуры (то есть активной области данных и дисплея) находится в регистре 1. Регистр 1 будет использоваться только для этой цели.

При генерации команд, в которых имеется обращение к переменным с индексами, мы полагаем, что это обращение выполняется с помощью информационного вектора, упоминавшегося выше. Предположим также, что элемент таблицы идентификаторов для имени массива C содержит адресную пару (номер области данных, смещение) информационного вектора, и будем считать, что в первой ячейке информационного вектора содержится CONSPART. Суть такого

размещения в том, что элементы массива на самом деле не будут размещаться в самой области данных – в ней мы размещаем информационный вектор. В нашем случае мы упростим представление – и считаем, что величина CONSPART для массива C будет равна значению переменной BASELOCK, которая представляет собой адрес первого элемента массива.

Программа, которую требуется сгенерировать для выражения B+P+C [M] +C[N], приведена на рис. 8.10. Для наглядности смещение для каждого обращения к переменной обозначается именем этой переменной.

```
LREG   2,0(1)   Занести адреса ОД главной программы
LREG   3,1(1)   и ОД для процедуры X в регистры 2, 3 для
                ссылки на B, P.
LOAD   B (2)    Загрузить B, используя регистр 2, содержащий
                адрес ОД главной программы.
ADD    P(3)     Прибавить P, находящееся в ОД процедуры X.
LREG   4,M (1)  Занести M в регистр 4.
LREG   5,C(1)   Занести CONSPART для массива C в
ADD    0 (5, 4) регистр 5.
                Регистры 4, 5 дают CONSPART+M, поэтому к
                (B+P) прибавляется C [M].
LREG   6, N (2) Для N – в ОД главной программы –
                используется регистр 2.
ADD    0 (5, 6) Регистр 5 содержит CONSPART для массива
                C, а регистр 6 содержит N. Наконец, прибавляем
                C [N].
```

Рис. 8.10. *Программа для выражения B+P+C[M]+C[N]*

Сначала выполняется обращение к B и P, поэтому две первые команды загружают адреса их областей данных в регистры 2 и 3. Две следующие команды выполняют сложение B+P. Затем для обращения к C [M] в регистр 4 заносится M, а в регистр 5 – значение CONSPART для массива C. Команда 7 прибавляет C [M] к (B+P). Далее для ссылки на индекс N необходим адрес области данных главной

программы, а он уже находится в регистре 2. Аналогично значение CONSPART для массива C уже находится в регистре 5.

Рассмотрим более реальное построение информационного вектора для нашего массива C [1:10].

Его на самом деле можно представить следующим образом:

0-я ячейка	1	10	10
1-я ячейка	1	CONSPART	
2-я ячейка	BASELOCK		

В нашем случае L_1=1, U_1=10, d1= U_1-L_1+1=10, n=1, CONSPART= BASELOCK-1, и адрес C[1] будет BASELOCK-1+1= BASELOCK, то есть BASELOCK указывает на расположение в памяти самого первого элемента массива C. Отметим, однако, что в вышеприведенном примере мы используем для простоты изложения другую форму адресации, располагая CONSPART непосредственно в области данных.

8.9. Более полная схема генерации кода для тетрад

Для формирования более подробного описания процесса генерации кодов нам необходима следующая информация о главных компонентах, а именно:

1. Формат описаний переменных в полях тетрад. Основные форматы представления уже были рассмотрены выше.
2. Формат описаний регистров и сумматора совместно с форматами применения этих описаний.
3. Формат адресов в готовой программе, которые заносятся в генерируемые команды для обращения к переменным при выполнении программы.
4. Процедуры для генерации команд загрузки значения в регистр и формирования адресной части команды.
5. Основной генератор команд для тетрады +.

8.9.1. Описания регистров и сумматора

Введем в рассмотрение массив XRVALUE, содержащий описания значений, которые будут находиться в сумматоре

и регистрах во время выполнения программы в тот момент, когда выполнится последняя сгенерированная команда.

XRVALUE (0) описывает сумматор, XRVALUE (i) описывает регистр i для i = 1, . . . , 7. XRVALUE (i) может содержать:

1. Код операнда (1-4), как обозначено в начале раздела 8.6. Это означает, что данный операнд находится в сумматоре или регистре.

2. Операнд (код – 5), который указывает, что регистр содержит значение CONSPART массива, описанного следующим образом:

5	0	указатель на элемент таблицы символов для имени массива

3. Операнд, указывающий, что в регистре находится адрес области данных:

6	0	№ области данных

На данный момент мы констатируем, что в регистрах (и соответственно, в XRVALUE(i)) могут находиться операнды 6-ти типов. Таким образом, нам известно, какое значение находилось в регистре, но нам необходимо еще знать, насколько важно, чтобы оно там и осталось. Например, регистр 1 вообще нельзя использовать в других целях, так как в нем всегда по умолчанию записан адрес активной области данных. Мы воспользуемся параллельным массивом XRSTATUS, элементы которого XRSTATUS(i) могут принимать следующие значения:

(0) Регистр i или сумматор пусты.

(1) Это значение есть еще и в памяти. Значит, нет необходимости запоминать регистр, если мы хотим использовать его для других целей.

(2) Этого значения нет в памяти.

(3) Содержимое регистра нельзя в данный момент изменять.

Начиная генерировать команды для тела процедуры, мы заносим 0 в XRSTATUS (J) для J= 0, 2,. . ., 7 и 3 в XRSTATUS (1). Кроме того, мы помещаем в XRVALUE (1) описание:

6	0	№ области данных процедуры

Таким образом, перед началом генерации кода для тетрады нам необходимы сама тетрада с операндами в соответствии с описанными выше форматами, две глобальные переменные INS1 и INS2, в которых будет формироваться текст ассемблерной команды, и массивы XRSTATUS и XRVALUE.

Описания сумматора и регистров заносятся каждый раз, когда из тетрады генерируются команды. В связи с этим используются две важные процедуры: FREEACC и FREEREG(I). Первая процедура проверяет описание сумматора XRVALUE(0) и XRSTATUS(0) и генерирует команды, если необходимо запомнить его содержимое:

1. Если XRSTATUS(0)=0, то выполнить возврат (сумматор свободен).
2. Если XRSTATUS(0)=1, сумматор содержит значение, находящееся также и в памяти. Занести 0 в XRSTATUS (0) и выполнить возврат.
3. Если XRSTATUS(0)=2, то этого значения нет в памяти. В нашей реализации это может случиться только тогда, когда XRVALUE (0) описывает временную переменную. Выделить для нее память в активной области данных, скажем, со смещением к. Сгенерировать команду ”STORE k(1)”. Занести 0 в XRSTATUS (0). Возврат.
4. Случай XRSTATUS (0) = 3 невозможен при нашей реализации.

FREEREG(I**)** выполняет аналогичные действия: если I≠0, то генерируются команды (если необходимо) для запоминания регистра I и изменения его описания на «свободен». Если I=0, то процедура может выбрать любой регистр, который она пожелает освободить, и выдать номер этого регистра в переменной I. Если программа выбирает регистр, то она в первую очередь должна отдать предпочтение регистру, который в данный момент свободен, во вторую очередь – регистру, содержимое которого уже находится в памяти (так что не надо генерировать команды), и в третью очередь – регистру, для которого XRSTATUS(I)=2. Как только регистр, который должен быть освобожден, определился, выполняются действия, такие же, как в процедуре FREEACC.

8.9.2. Ассемблерные команды адресации операндов

При генерации кода для операции, такой, как (+A, B, T1), мы должны знать, по какому адресу следует обращаться к A и B, находящимся в памяти.

В таблице, приведенной ниже, перечислены различные операнды, которые могут встретиться во внутреннем представлении исходной программы. Во второй колонке приведены команды, которые можно сгенерировать, чтобы определить адрес операнда, тогда как в третьей колонке даются адрес и номер индексного регистра в команде, которая в конечном счете обращается к операнду. При необходимости может быть установлен бит косвенной адресации. Первые две строки таблицы на самом деле не являются ссылками на ячейки памяти, но мы таким образом указываем, что операнд находится в сумматоре или в регистре. Мы можем это сделать, поскольку рассматривается такой случай, когда в ссылке на память используется по меньшей мере один регистр.

Операнд	**Команды ассемблера**	**INSTR**
В сумматоре		0(0,0)
В регистре j		j(0, 0)
Ячейка в активной области данных, смещение к		к(1)
Ячейка в другой области данных, смещение к	LREG m, адрес DA	k(m)
Переменная с индексами в активной области данных. CONSPART имеет смещение к	LREG j,индекс LREG n,k(l)	0(n, j)
Переменная с индексами в другой области данных. CONSPART имеет смещение k	LREG j,индекс LREG m,адрес DA LREG n,к(m)	0(n, j)

В команде для обращения к ячейке памяти используется абсолютное целое число к (смещение), два индексных регистра и бит для указания косвенной адресации. Обозначение k (i, j) будет использоваться как ссылка на ячейку с адресом:

содержание к + содержимое регистра I + содержимое регистра j

Будем считать, что если i равно 0, то «содержимое регистра i» также равно 0. Другими словами, к (0, 0) ссылается на ячейку к. Бит косвенной адресации отмечается звездочкой: *k (i, j) ссылается на ячейку, адрес которой находится в ячейке k (i, j). Конечный адрес образуется как результат сложения смещения с содержимым значений в регистрах.

8.9.3. Список переменных и процедур генерации кода

Четыре поля тетрады i:

QD (i).OP, QD(i).OPER1, QD(i).OPER2 и QD (i).RESULT.

Две глобальные ячейки INS1 и INS2, используемые для хранения адресов операндов, представленных в текстовом виде кода ассемблера.

Два массива XRVALUE и XRSTATUS используются для описания текущего содержимого сумматора и регистров; в XRVALUE (0) описано содержание сумматора, в XRVALUE (i) описано содержание регистра i.

Процедура **FREEACC()** генерирует команды (если это необходимо) для запоминания содержимого сумматора и меняет описание сумматора на «свободно».

Процедура **GETINREG (OPERAND, I)** генерирует команды для загрузки OPERAND в какой-либо регистр и выдает в I номер этого регистра. OPERAND не должен быть описанием переменной с индексами.

Процедура **GETINACC (OP1, ADD1, OP2, ADD2),** как нам известно, генерирует (если необходимо) команду загрузки одного из операндов OP1 или OP2 в сумматор. OP1 – операнд, ADD1 – соответствующий ему адрес в ассемблерной команде (задается смещение, номера одного или двух регистров и бит косвенной адресации). Если OP2 уже находится в сумматоре, операнды меняются местами.

Процедура **FIXAD (OPERAND, INSTR)** заносит в INSTR адрес (смещение, номера одного или двух регистров и бит

косвенной адресации) для ссылки на OPERAND. Процедура выдает (0,0), если OPERAND уже находится в сумматоре, и j (0,0) – если он в регистре j.

Процедура **FIXADMEMORY (OPERAND, INSTR)** подобна процедуре FIXAD. Единственное отличие состоит в том, что она должна выдать в INSTR адрес ячейки памяти. Следовательно, если OPERAND находится в сумматоре или в регистре, но не в памяти, процедура должна сгенерировать команды для запоминания OPERAND в памяти и выдать его адрес.

Процедура GETINREG (OPERAND, I)

Из приведенной выше таблицы ясно, что нам придется время от времени генерировать команды загрузки в регистры индексов значений CONSPART для массивов и адресов областей данных. Данная процедура делает это для OPERAND и заносит в переменную I номер регистра. Единственное ограничение состоит в том, что OPERAND не может быть переменной с индексами. В процедуре используются локальные переменные j, J, D, OP и к.

1. Если OPERAND находится в регистре j, занести j в I и выполнить возврат (при проверке сравнивается OPERAND с XRVALUE 0) для j = 1, ..., 7) – по условию у нас только 7 регистров.

2. Если OPERAND находится в сумматоре, выполнить следующее: I:=0; вызвать FREEREG(I); сгенерировать команду «LRACC I». Переписать XRSTATUS(0) и XRVALUE(0) в XRSTATUS(I) и XRVALUE(I). Возврат.

3. Если OPERAND имеет вид (6, 0, D), то D – номер области данных, и нужно занести ее адрес в регистр. Выполнить следующие действия: I:=0; вызвать FREEREG(I); сгенерировать LREG I, к (1), где к – смещение в активной области данных для ссылки на адрес области данных D. Занести 1 в XRSTATUS(I) и OPERAND в XRVALUE(I). Возврат.

4. OPERAND – константа, временная переменная или идентификатор, описанный в некоторой таблице. Пусть D – номер области данных операнда, к – его смещение. Сформировать операнд OP:=(6,0,D) и вызвать GETINREG (OP, J).

(В результате адрес области данных окажется в регистре J. Заметьте, что это рекурсивный вызов.) Установить I:=0; вызвать FREEREG(I); сгенерировать «LREG I,k(J)» или «LREG I,*k(J)», в зависимости от ее состояния бита косвенной адресации в OPERAND. Занести 1 в XRSTATUS (I) и OPERAND в XRVALUE (I). Возврат.

Процедура FIXAD(OPERAND, INSTR)

Для любого операнда из некоторой тетрады эта процедура формирует адресную часть команды, которая ссылается на него, и заносит этот адрес в INSTR. Результат в INSTR может также иметь вид 0 (0,0) или j (0,0), если OPERAND уже находится в сумматоре или в регистре j:

1) Если OPERAND в сумматоре, занести адрес (0,0) в INSTR; возврат.

2) Если OPERAND в регистре j, занести адрес j(0,0) в INSTR; возврат.

3) Если OPERAND описывает переменную с индексами, выполнить следующие шаги:

a) Пусть SUBSCRIPT – та часть OPERAND, которая описывает индекс. Вызвать GETINREG (SUBSCRIPT, I).

Занести адрес 0(0,I) в INSTR.

b) Пусть P – указатель на элемент таблицы символов для имени массива. Сформировать операнд OP:=(5,0,P). Установить I:=0. Вызвать GETINREG(OP, I), чтобы сгенерировать команды занесения CONSPART в регистр. Приформировать адрес 0(I ,0) в INSTR.

c) Установить бит косвенной адресации в INSTR, если это требует OPERAND. Возврат.

4) OPERAND – константа, временная переменная или переменная, для которой имеется элемент в таблице символов. Значение не находится ни в сумматоре, ни в регистре. Выполнить следующие шаги:

а) Пусть D – номер области данных адреса операнда. Сформировать операнд OP:=(6,0,D). Вызвать GETINREG (OP, I).

б) Пусть к – смещение для OPERAND в области данных. Занести адрес k(1) в INSTR. Установить бит косвенной адресации в INSTR, если это требует OPERAND. Возврат.

8.9.4. Генерация кода для арифметических тетрад

Теперь мы готовы в общих чертах описать процедуру генерации команд для i-й тетрады вида (+ A,B,T1). Генерация команд для других тетрад аналогична. Первая задача состоит в генерации адресов двух операндов и занесении этих адресов в две глобальные ячейки INS1 и INS2. Далее следует упрощенная схема генерации команд для ADD.

1. Установить INS1:=0; INS2:=0 (инициализация адресов операндов).

2. Вызвать FIXAD (QD(i).OPER1, INS1). (Эта процедура генерирует команды, если это необходимо, которые вычислят адрес OPERAND 1 во время выполнения программы; этот адрес в виде k (i,j) заносится в INS1.)

3. Вызвать FIXAD(QD(i).OPER2, INS2).

4. GETINACC(QD(i).OPER1, INS1,QD(i).OPER2, INS2).

Цель состоит в генерации команд (если это необходимо) занесения одного из операндов в сумматор. Каждый операнд представлен двумя значениями – собственно описанием OPERAND и его адресом в готовой программе. При выходе из процедуры первый операнд будет в сумматоре.

5. Выполнить этот шаг только в случае, если второй операнд находится в сумматоре или в регистре. Мы хотим сгенерировать команду ADD, а для этого нужно, чтобы операнд находился в памяти. Если INS2=0(0,0) или j(0, 0), мы должны повторить вычисление адреса второго операнда, используя для этого иную процедуру; INS2:=0; вызвать FIXADMEMORY(QD(i).OPER2, INS2).

6. Сгенерировать команду ADD с адресом INS2. Изменить описание сумматора, чтобы указать, что в нем содержится QD(i).RESULT (занести 2 в XRSTATUS(0)).

7. Если какой-либо из операндов QD(i).OPER1 или QD(i).OPER2 является временной переменной, зона существования которой ограничена тетрадой i, выполнить следующее: если для операнда была выделена память, то освободить ее. Если операнд находится в регистре, в описании этого регистра отметить, что он свободен.

8. INS1:=0; INS2:=0.

Две ячейки: INS1 и INS2 – глобальные, и их следует использовать во всех программных сегментах, которые генерируют команды для бинарного или унарного оператора, по следующим причинам. Предположим, что мы генерируем адрес первого операнда и что он использует регистры 2 и 3. Тогда при генерации адреса второго операнда важно, чтобы регистры 2 и 3 оставались нетронутыми, так как они содержат величины, необходимые для обращения к первому операнду. Следовательно, FREEREG должна сама сохранить содержимое этих регистров. В процедуру FREEREG нужно внести упоминавшиеся при ее описании дополнительные ограничения:

если либо INS1, либо INS2 содержат ссылки на регистр j в виде j(0,0), k(j,m) или к(m,j), этот регистр не должен освобождаться.

Хотя идея генерации команд для тетрад довольно прозрачна, генерация даже локально эффективной программы может оказаться достаточно сложной. В каждой процедуре предусматривается несколько особых подслучаев, и каждый из них требует индивидуальной обработки. Отладка может никогда не закончиться, так как всегда остается один последний особый случай, о котором забыли. Некоторые случаи требуют особого внимания: 1) оба операнда при сложении одновременно находятся в сумматоре (например, при оптимизации A*B+A*B), 2) значение одновременно находится в сумматоре и регистре (например, при сложении M+A[M]).

Ясно, что процесс формирования адресов операндов следует тщательно продумать и несколько раз смоделировать вручную до начала реализации.

8.9.5. Другие методы генерации кодов

8.9.5.1. Создание общих подпрограмм

Общий план генерации команд для любого арифметического или логического оператора состоит в том, чтобы:

сгенерировать команды для получения адресов операндов;

сгенерировать команды для выполнения необходимых преобразований типов;

сгенерировать команды для загрузки одного из операндов в сумматор и, наконец,

сгенерировать команды для операции.

Во многих случаях можно написать одну общую процедуру генерации команд, пригодную для всех арифметических и логических операторов. Параметрами для нее будут: описания операндов, требуемый тип результата, указание о коммутативности операции, указание о существовании обратной операции (например, обратное деление) и некоторые другие сведения об особенностях конкретной машины. Важным параметром является последовательность каркасных команд; она используется процедурой генерации команд (см. предыдущий раздел) на шаге 4. Эта же идея используется в компиляторе IBM FORTRAN H (хотя там схема генерации команд несколько отличается).

8.9.5.2. Шкалы

В широко известном оптимизирующем компиляторе IBM FORTRAN H весьма искусно решен вопрос упаковки таблиц, содержащих информацию о том, как генерировать команды. Чтобы проиллюстрировать это, предположим, что каждый регистр можно также использовать как арифметический регистр (как в родоначальнике всех IBM-мовских машин IBM 360).

Проход, предшествующий генерации, обрабатывает тетрады и тем самым осуществляет довольно сложную глобальную оптимизацию регистров. Для каждой тетрады (например, для +I, T1,T2, T3, где +I обозначает сложение целых величин) известно, какие из операндов уже находятся в регистрах, какие значения операндов должны оставаться в этих регистрах и т.д. Известен регистр Ri (i = 1, 2, 3), выделенный для хранения значения Ti, если это необходимо, а в регистре Bi должен храниться адрес области данных, где находится Ti. (Очевидно, если Ti в активной области данных, то Bi = 1.) Эта информация находится в тетраде вместе с операндами. Некоторые из регистров могут совпадать.

Например, если нет необходимости оставлять T1 в регистре, полагаем R3 = R1.

Кроме того, наряду с тетрадой существует 8-разрядное поле STATUS, содержащее следующую информацию (если разряд содержит 1, то информация истинная, в противном случае справедливо обратное):

№ Разряда	Смысл
1	Операнд T1 уже в регистре R1
2	Значение операнда T1 должно оставаться в регистре R1
3	Операнд T2 уже в регистре R2
4	Значение T2 должно оставаться в регистре T2
5	Адрес области данных, содержащей T1, еще не в регистре B1
6	Адрес области данных, содержащей T2, еще не в регистре B2
7	Адрес области данных, содержащей T3, еще не в регистре B3
8	После операции T3 должно быть занесено в память

Команды, которые могут быть сгенерированы для конкретной операции, хранятся в таблице в форме каркаса в том порядке, в котором они будут генерироваться, с каждой командой связана некоторая последовательность битов. Это иллюстрируется на рис. 8.11 для оператора +I. Процесс генерации команд происходит так: общей программе генерации передается тетрада (вместе со всей дополнительной информацией) и таблица команд для соответствующего оператора. Программа анализирует таблицу следующим образом:

Первые четыре разряда 8-разрядного поля STATUS используются для выбора столбца разрядов состояния: 0000 – выбирается первый столбец, 0001 – второй и т.д.

В выбранном столбце все x-разряды заменяются последними четырьмя разрядами из поля STATUS. Последний x заменяется восьмым разрядом, предпоследний – седьмым и т.д.

Последовательно просматривается столбец разрядов, и если разряд содержит 1, генерируется соответствующая команда.

Чтобы показать, как это делается, рассмотрим 8-разрядное поле 00000011, указывающее, что ни T1, ни T2 не находятся в регистрах, ни один из операндов не нужно оставлять в регистре, регистры B1 и B2 уже загружены, B3 еще не загружен, и результат следует запомнить.

Первые четыре разряда выбирают столбец 1, который имеет вид x010x010xx. Заменяя все x последними четырьмя разрядами, получаем 0010001011

Каркасные команды	Разряды состояния – содержание 4-х разрядов Status'a
	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
LREG B1,D(1)	x x x x x x x x 0 0 0 0 0 0 0 0
LREG R1,D(B1)	0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0
LREG R3,D(B1)	1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
MREG R3,R1	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
LREG B2,D(1)	x x x x x x x x x x 0 0 x x 0 0
LREG R2,D(B2)	0 1 0 0 0 1 0 0 0 1 0 0 x x 0 0
ADD R3,D(B2)	1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0
ADDR R3,R2	0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1
LREG B3,D(1)	x x x x x x x x x x x x x x x x
STORE R3,D(B2)	x x x x x x x x x x x x x x x x

Рис. 8.11. *Шкалы для оператора +I*

Следуя каркасам команд, мы генерируем (по порядку) третью, седьмую, девятую и десятую команды:

LREG	R3,D(B1)	Загрузить T1 в R3.
ADD	R3,D (B2)	Прибавить T2 к R3.
LREG	B3,D (1)	Подготовить адрес T3.
STORE	R3,D (B3)	Запомнить результат.

Смещения, обозначенные буквой D, определяются из описаний операндов, а точнее – из областей данных, где они располагаются; номера регистров заданы в тетрадах.

Сгенерированную программу можно сделать более эффективной, если внести в каркас дополнительные команды, в которых учитываются коммутативность некоторых операций и другие особенности.

8.10. Контрольные вопросы по теме «Генерация кодов»

1. Что такое «переменная времени компиляции»?
2. Что представляет собой информация в переменной в переменной ACC?
3. Какие действия выполняет процедура GETINACC?
4. Как можно генерировать код для троек?
5. Что такое «Зона существования временной переменной»?
6. В чем смысл генерации кода для троек в виде дерева?
7. Каково предназначение процедуры COMP()?
8. Почему для троек нужны таблицы процедур генерации?
9. За счет чего можно добиться генерации более оптимального кода?
10. Векторы и матрицы. Постоянная величина CONSPART.
11. Что такое «информационный вектор»?
12. Что такое «Область данных»?
13. В чем различие статического и динамического выделения памяти под переменные?
14. Что такое дисплей и где его можно хранить?
15. Сумматор и регистры. Отслеживание информации в этих устройствах.
16. Опишите прицип генерации с помощью шкал.

9. МАШИННО-ЗАВИСИМАЯ ОПТИМИЗАЦИЯ

После генерации кодов возможно приступить к машинно-независимой оптимизации, которая предусматривает в основном максимальное использование специфичных характеристик конкретной архитектуры, например RISC или CISC, таких как регистры общего назначения или специальные регистры. Как правило, общего количества регистров всегда не хватает для создания качественного и эффективного кода.

Рассмотрим два основных существующих ныне типа компьютерных архитектур.

CISC (complex instruction set computer) – компьютер с полным набором инструкций..

RISC (reduced instruction set computer) – компьютер с сокращенным набором инструкций.

Архитектуры CISC разрабатываются с прицелом на последующую реализацию языков высокого уровня, следовательно, на первый взгляд являются идеальными кандидатами для использования в компиляторах. Они имеют множество мощных инструкций и ассоциируются с компактным целевым кодом. Их типичными характеристиками являются:

- широкий диапазон режимов адресации для обеспечения доступа к массивам, записям, спискам, стековым фреймам и т.д.;
- небольшое число регистров (обычно 16 или меньше);
- большое число регистров особого назначения, например, может быть выделен регистр для индексации;
- двухадресные инструкции, такие как A+B→A, где A, B могут быть сложными адресами;
- инструкции переменной длины;
- инструкции с побочными эффектами, например, команды автоматического приращения;
- существенно разное время, затрачиваемое на выполнение инструкций;
- управление, реализованное микропрограммой.

С другой стороны, RISC имеет следующие характеристики:

- простые режимы адресации (обычно – только использующие регистры);
- большое количество регистров, по меньшей мере – 32;
- все регистры являются «универсальными»;
- трехадресные инструкции, использующие только регистры, например: r3 = r1 + r2;
- инструкции фиксированной длины (32 бита);
- отсутствие у инструкций побочных эффектов;
- примерно одинаковое время, затрачиваемое на выполнение любой инструкции;
- более сложное управление по сравнению с микропрограммным.

Архитектуры RISC выигрывают по сравнению с CISC с точки зрения простоты и наличия сравнительно меньшего числа способов достижения необходимого результата (меньше вариантов для вычислений и альтернатив выбора в процессе компиляции). К числу преимуществ относятся также инструкции с фиксированной длиной и наличие большого числа универсальных регистров. Отметим, что архитектура целевого компьютера – это самый важный вопрос при выборе стратегии генерации кода.

9.1. Распределение регистров

При создании качественного целевого кода критичным процессом является распределение регистров. Как правило, в общем случае операции с использованием содержимого регистров занимают меньше времени по сравнению с соответствующими операциями с использованием значений, находящихся в основном пространстве памяти. Это означает, что для операндов команд объектного кода необходимо (насколько это возможно) использовать именно регистры. Существуют три основных типа значений, которые нужно помещать в регистры при любой возможности.

Часто используемые указатели на структуры данных времени выполнения, например, на стек времени выполнения.

Значения параметров функций и процедур.

Значения временных переменных, которые применяются при вычислении выражений.

Для архитектуры RISC не составит проблемы использовать регистры для указателей на стек времени выполнения, поскольку в этой архитектуре имеется большое количество регистров, и для указанных целей может свободно выделяться блок регистров.

Для архитектуры CISC, где количество регистров ограничено, такое использование регистров не всегда будет возможным. Обычным решением данной проблемы для архитектуры CISC является использование регистров только для наиболее часто употребляемых указателей на стек времени выполнения, например, указателей на основание стека, указателей на текущий стековый фрейм и указателей на вершину стека. Благодаря этому доступ к локальным переменным, чьи значения содержатся в текущем стековом фрейме, становится более эффективным по сравнению с доступом к переменным, чьи значения содержатся в других областях. Это обстоятельство может быть использовано программистами, знающими подробности реализации.

Конечно, даже в архитектуре RISC может случиться так, что блока регистров, выделенного для указателя кадра, окажется недостаточно (вследствие динамической глубины вложения функций и процедур). Обойти проблему нельзя никак, поскольку в наличии имеется только ограниченное число регистров, и, если их недостаточно, значения из одного или большего числа регистров нужно будет сбросить (spill) в область памяти, отличную от выделенной под регистры.

Машинно-зависимая оптимизация кода включает в себя:

а) Назначение и распределение регистров.

Инструкции, использующие регистры, выполняются быстрее, чем инструкции, требующие обращения к памяти. Значит, нужно стараться разместить на регистрах все переменные и промежуточные результаты, которые потом используются в программе.

Но редко бывает доступно столько регистров, сколько нужно в данный момент. Значит, нужно выбрать регистр,

переменная в котором должна быть заменена при необходимости использовать этот регистр для другой цели.

Для этого выполняется просмотр программы вперед, чтобы определить, когда каждый из регистров будет повторно использоваться. Переменная, которая не потребуется в течение наибольшего времени, как раз и определит искомый регистр.

б) Учет операторов передачи управления.

Для этого программу разбивают на линейные участки, в пределах которых мы выполняли ранее МНО.

Началом для нового линейного участка может быть: каждая команда, на которую может быть передано управление в результате операции перехода; или команда, непосредственно следующая за командой передачи управления; или операция вызова подпрограммы.

Внутри линейного участка регистры распределяются вышеописанным способом. При переходе к новому участку все значения регистров сохраняются в рабочих переменных.

в) Изменение порядка выполнения команд с целью сокращения операций обращения к памяти.

г) Использование специфических характеристик и инструкций конкретной машины. Например, специальных инструкций организации циклов или соответствующих доступных способов адресации.

Наиболее известны в настоящее время пять основных приемов оптимизации применения регистров.

Локальная оптимизация регистров

Основная идея – по возможности хранить результаты исполнения тетрад (троек и т.д.) в регистре, не используя его в тех тетрадах, в которых есть ссылки на этот регистр. Пересылка содержимого регистра в память осуществляется лишь тогда, когда свободные регистры отсутствуют. Для такого хранения результатов выделяется специальная группа регистров, называемых локальными регистрами.

Глобальная оптимизация регистров

Это выглядит как попытка обеспечить хранение наиболее часто использующихся значений переменных или констант или на специально выделенных для этой цели регистрах, или на имеющихся свободных регистрах. Решение о выделении

регистра для такой цели принимается на основании счетчика ссылок на ту или иную величину. Особое внимание уделяется использованию регистров внутри циклов, а также в командах, где используются два или три регистра (например: команда BXLE в коде IBM-ассемблера.)

Перераспределение глобальных регистров

После этих двух вышеописанных оптимизаций может оказаться, что некоторым переменным выделены как локальные, так и глобальные регистры, и как следствие – часто применялись инструкции перезагрузки регистров. В связи с этим возникает необходимость в третьем шаге.

Утилизация локальных регистров

После перераспределения регистров некоторые локальные регистры могут оказаться свободными, так как хранившиеся в них переменные окажутся переведенными на глобальные. В этом случае освободившиеся локальные регистры будут хранить адресные константы, для которых ранее не было места в регистрах.

Чистка глобальных регистров завершает оптимизацию.

Подобная идеология использования локальных и глобальных регистров реализована в виртуальной машине MMIX.

10. РАСПРЕДЕЛЕНИЕ ПАМЯТИ

Этап синтеза исходной программы тесно связан с генерацией кода, а важным и родственным этой теме вопросом является распределение памяти. Связь между генерацией кода и распределением памяти следующая: распределение памяти обычно рассматривают как отдельную фазу процесса компиляции, которую, по необходимости, вызывает генератор кода. Вот почему мы проблему распределения памяти рассматриваем непосредственно после кодогенерации. В данной главе будут рассмотрены следующие вопросы.

- Типы объектов, для которых нужно выделять память.
- Влияние ожидаемого времени существования объекта на механизм выделения для него памяти.
- Распределение памяти для конкретных языковых характеристик.
- Основные используемые модели распределения памяти.

10.1. Память

Для начала обсуждения следует определить отличие объектов от их значений. Переменная *x* является объектом, который в данное время может иметь соотнесенное с ним значение, занимающее определенный объем памяти. Память, выделенная значению *x*, имеет *адрес*, который позволяет обращаться к *x*, причем адрес должен иметь следующие свойства.

- Быть достаточно (но не слишком) большим, чтобы вместить любое из значений, которое может принимать x.
- Быть доступным в течение всего времени существования x.
- Должна существовать возможность его выражения в такой форме, чтобы генератор кода мог пользоваться адресом для получения доступа к значению x во время выполнения программы.

Относительно первого требования следует сказать, что целые значения обычно занимают меньше памяти, чем действительные, а символьные значения могут занимать меньше

памяти, чем целые. В то же время для обеспечения эффективного доступа не всегда имеет смысл уплотнять в памяти значения до максимально возможной степени (в любом случае экономия получается мизерная). Отметим также, что некоторые языки позволяют пользователю определить, требуется ли уплотнять значения.

Память также требуется для значений записей, массивов и указателей. Память, требуемая для записей, обычно равна сумме объемов памяти, требуемых для каждого поля записи. Для массивов требуется больше памяти, чем для составляющих их элементов; избыток зависит от способа хранения массива. Кроме того, некоторые языки допускают наличие у массивов *динамических границ*, следовательно, в процессе компиляции размер массива неизвестен и будет определен уже во время выполнения программы. Объем памяти, необходимой для указателей, зависит от реализации.

В связи с тем, что память выделяется на все время жизни переменной, возможны следующие ситуации:

- Время жизни переменной равно времени жизни программы. В этом случае выделенная для переменной область памяти уже не может быть освобождена. Такую память называют *статической*.
- Переменная объявляется в каком-то конкретном блоке, в какой-то функции или процедуре. В этом случае после завершения выполнения блока, функции или процедуры выделенную для переменной память можно освободить. Такую память называют *динамической*. Именно такую память мы использовали в предыдущих разделах.
- Память может выделяться значениям, не обязательно соотнесенным с переменными, в определенный момент выполнения программы, не обязательно совпадающий с началом блока или входом процедуры. Таким образом, память выделяется в этот момент времени и существует до тех пор, пока не будет освобождена – либо посредством соответствующего механизма языка, либо после того, как просто станет недоступной для программы. В то же время сам момент освобождения памяти в общем случае может не определяться

при компиляции, а станет известным только во время выполнения программы. Такую память называют *глобальной*.

Требования к статической памяти полностью определяются во время компиляции, так что необходимый объем может быть выделен. Поскольку выделенную статическую память освободить невозможно, общий объем такой памяти является суммой ее частных составляющих, при этом какое-либо «совместное использование» этой памяти невозможно. Управление статической памятью является простым. К примеру, для программ на языке Фортран все требования к памяти являются статическими.

Требования к динамической памяти программы сложнее, поскольку память распределяется на входе функции (блока или процедуры в зависимости от рассматриваемого языка), а освобождается после выполнения функции (блока или процедуры). В этом случае существует возможность совместного использования этой памяти значениями, относящимися к различным функциям и т.д. Оказывается, что управление этим типом памяти не настолько сложно, как может показаться на первый взгляд, и его легко осуществить посредством механизма стека, который увеличивается и уменьшается при выделении и освобождении памяти. Несколько подробнее данный вопрос будет рассмотрен ниже.

Распределение глобальной памяти осуществляется достаточно просто: область пространства (обычно называемая *кучей*) увеличивается настолько, насколько это необходимо. *Освобождение* этой области памяти осуществляется намного сложнее, поскольку данный процесс трудно связать с процессом распределения памяти. Существует два основных вопроса, связанных с распределением и освобождением глобальной памяти.

- Доступность памяти для освобождения определяется во время выполнения программы, что неизбежно приводит к некоторого рода служебным издержкам при выполнении программы.
- После освобождения некоторого участка памяти в куче возникают чистые участки, которые обычно

требуют *сжатия* для более эффективного использования памяти.

На данный же момент просто отметим, что стек и куча могут удобно сосуществовать вместе, если их увеличение происходит по направлению друг к другу (рис. 10.1). В этом случае область статической памяти может размещаться на одном или другом конце пространства памяти, как это изображено на рисунке. Вмешательство извне потребуется только в том случае, когда взаимное расширение стека и кучи приведет к их «встрече», то есть нехватке памяти. Обычно в подобном случае определяется недоступное пространство кучи и происходит ее сжатие.

Рис. 10.1. *Общий вид памяти программы*

При рассмотрении адресов переменных и т.д. следует отметить, что иногда (например, при использовании статической памяти) адреса времени выполнения известны во время компиляции. В то же время чаще имеем обратную ситуацию, когда адреса времени выполнения должны вычисляться, исходя из множества факторов, часть которых известна в процессе компиляции, а часть неизвестна до начала выполнения программы. В этом случае аспекты адреса, известные при компиляции, называются *адресом времени компиляции*.

В языке С для переменных имеется четыре возможных класса памяти: *static*, *auto*, *extern* и *register*. Для *статических* переменных память выделяется на все время программы. Для переменных класса *auto* (класс по умолчанию) память выделена для момента завершения работы составного оператора (термин языка С соответствует *блоку* в некоторых других языках), в котором были объявлены такие переменные. Таким образом, более удобной для этих переменных является память в стеке. Для переменных класса *extern* память выделяется в другом файле. Значения переменных класса *register* хранятся в регистре,

если компилятор способен организовать это удобным образом, в противном случае такие переменные эквивалентны переменным *auto*.

Помимо памяти, необходимой переменным программы на С, с помощью известной функции malloc() можно выделить память для значений, к которым обращаются посредством указателей, например:

P = malloc(sizeof(int));

Этой инструкцией выделяется достаточно памяти для целого значения, а P является указателем на это значение. Эта память *может* освобождаться после того, как ни одна переменная программы (в том числе P) не будет указывать на данную область памяти. В то же время, поскольку это невозможно определить в процессе компиляции, область, выделяемая посредством функции malloc(), обязательно должна располагаться в куче.

10.2. Статическая и динамическая память

Ранние языки программирования, такие как Фортран, имели статическую память, размер которой был известен во время компиляции. Выделенный объем памяти уже не освобождался, поэтому применялась очень простая модель распределения памяти – необходимая память выделялась от одного края доступного пространства по направлению к другому. Более современные языки, начиная с Алгол 60, обычно имеют блочную структуру, что позволяет переменным, объявленным в различных блоках, совместно использовать одну область памяти. Таким образом, удобными являются основанные на стеках модели распределения памяти, которые позволяют повторно использовать ранее выделенную память. Используемый в этих моделях стек времени выполнения в некотором смысле подобен таблице символов, но с одним важным отличием: стек времени выполнения – это структура *времени выполнения* программы, а не времени ее компиляции.

10.3. Некоторые приемы выделения памяти

10.3.1. Присваивание адресов переменным

Как уже было сказано, для хранения значений величин разных типов применяются две основные формы представления памяти – статическая и динамическая.

Мы исходим из того, что для каждой компилируемой процедуры компилятор отводит одну область данных. В эту область входят все неявные параметры, фактические параметры, переменные, определяемые программистом (или их описатели), и временные переменные, используемые процедурой.

В некоторых языках одновременно может компилироваться только одна подпрограмма (процедура), и работа ведется только с одной областью данных. В противоположность этому, например, в Алголе или PL/I, одновременно может компилироваться любое количество процедур, и каждая со своей областью данных.

В общем случае в компиляторе формируется некоторая таблица D, где перечислены все области данных объектной программы (все ОД). Элемент таблицы D для каждой области данных содержит поле, в котором помечено, статическая эта область или динамическая, поле LENGTH, указывающее на число ячеек, занимаемых областью, и т.д. Сначала LENGTH=0. Процедура распределения памяти, осуществляющая присвоение переменным адресов, каждый раз имеет дело с одной процедурой. Ее первая задача – отвести в начале области место для дисплея (если он используется) и для неявных параметров. В поле LENGTH устанавливается количество ячеек, используемых этими стандартными объектами фиксированного размера.

Вторая задача состоит в том, чтобы обработать элементы таблицы символов, соответствующие формальным параметрам, а также переменным, описанным в этой процедуре, и присвоить им относительные адреса.

Для каждого элемента таблицы символов делается следующее:

- Элементу таблицы символов присваивается смещение, равное значению LENGTH, то есть ему присваивается адрес первой свободной ячейки области данных.
- LENGTH увеличивается на число ячеек, отводимых под формальный параметр или переменную, описываемую этим элементом таблицы символов.

Это простая и очевидная обработка, нужно только, чтобы из таблицы символов можно было узнать количество ячеек, требуемых для каждой переменной. В некоторых языках (например, в Фортране IV) эта информация не известна до окончания просмотра всей исходной программы. Поэтому для присваивания адресов после обычного анализа необходим второй просмотр.

В языках, которые требуют описания переменных до их использования, распределение памяти может выполняться семантическими программами, предназначенными для обработки описателей. В этом случае действительно возможен однопроходный компилятор.

10.3.2. Переменные с начальными значениями

До сих пор мы полагали, что на этапе компиляции нам не нужна область данных процедуры. Как правило, это действительно так; нам нужен только счетчик LENGTH, указывающий на размер области данных. В большинстве языков имеется возможность задавать переменным начальные значения, которые присваиваются во время компиляции. Самый легкий способ осуществить это – создать область данных с начальными значениями в виде части объектной программы и занести в нее начальные значения. Другое решение (вероятно, не столь хорошее) – сгенерировать последовательность операторов присваивания для установки начальных значений.

10.3.3. Проблемы выравнивания

Для IBM 360 требуется, чтобы некоторые величины начинались с границы двойного слова, слова или полуслова. Предположим, нам нужно отвести память переменным B1, F1, H1 и D1 в том порядке, как они написаны (с выравниванием,

если это необходимо), причем каждая из этих переменных занимает соответственно 1, 4, 2 и 8 байтов, и их адреса должны быть кратны 1, 4, 2 и 8 соответственно. Здесь плохо то, что многие байты совсем не используются. Чтобы избежать этого, можно распределять память сначала для переменных, которые должны начинаться с границы двойного слова, затем с границы слова, полуслова и байта.

10.3.4. Отведение памяти временным переменным

Временные переменные нужны главным образом для хранения промежуточных результатов вычисления выражений. Они нужны также для хранения адресов переменных с индексами, значений фактических параметров и списков параметров при обращении к функциям. Обычно для временных переменных отводятся ячейки в той области данных процедуры (или блока), в которой встретилось выражение. Мы могли бы для каждой временной переменной выделить отдельную ячейку, но это было бы неэкономно. Поэтому нам хотелось бы иметь такой алгоритм распределения памяти, который бы минимизировал количество используемых ячеек. Для начала мы будем рассматривать только линейные участки, то есть последовательность команд, выполняемых по порядку и не имеющих переходов ни внутрь, ни наружу.

Рассмотрим последовательность тетрад для следующих инструкций:

E:=A*B+C*D; F:=A*B+1; G := C*D;

```
(1)    * A, B, T1      (4)  := T3, , E    (7)  := T5, , F
(2)    * C, D, T2      (5)  * A, B, T4    (8)  * C, D, T6
(3)    + T1,T2,T3      (6)  + T4, 1, T5   (9)  := T6, , G
```

Поскольку нас интересуют одни только ЗАПиси и ЧТения временных переменных T1, T2, ..., это можно записать так:

```
(1)  ЗАП T1      (5)  ЗАП T3      (9)  ЗАП T5
(2)  ЗАП T2      (6)  ЧТ T3       (10) ЧТ T5
(3)  ЧТ T2       (7)  ЗАП T4      (11) ЗАП T6
(4)  ЧТ T1       (8)  ЧТ T4       (12) ЧТ T6
```

Зоной существования (или временем существования) временной переменной Ti мы назовем последовательность операций от начальной установки до ее последнего использования. Смысл этой зоны – вовремя распознать освобождение той или иной переменной. Очевидно, временные переменные с непересекающимися зонами могут располагаться в одних и тех же ячейках. В приведенном примере T1, T3, T4, T5 и T6 можно последовательно располагать в одной ячейке.

Часто алгоритм распределения памяти для временных переменных работает после того, как команды машины уже сгенерированы. Он имеет дело с командами машины, а не с внутренним представлением. Это делается по той причине, что не для всех временных переменных требуются ячейки памяти.

Например, значение T5 в командах машины, вероятно, будет оставаться на сумматоре или в быстром регистре на протяжении всей своей зоны и не потребует никакой ячейки памяти. Но об этом известно только тогда, когда команды уже сгенерированы.

Алгоритм может быть включен в фазу генерации программы или работать как отдельный проход после генерации программы распределения памяти в виде отдельного прохода в терминах последовательности ЗАПисей и ЧТений.

10.3.5. Указание о зоне в описателе временной переменной

Информацию о зонах можно хранить в описателях временных переменных в таблице символов. Мы укажем три способа хранения:

1) В описателе временной переменной содержится счетчик числа, сколько раз эта временная переменная встречается в тетрадах (он увеличивается при появлении новых тетрад, использующих данную временную переменную, и корректируется в процессе оптимизации программы). При генерации команд по тетраде счетчик уменьшается на 1 для тех временных переменных, которые встречаются в поле операнда или в поле результата этой тетрады. Если счетчик равен нулю, то данная

временная переменная больше не нужна и место, отведенное для нее, считается свободным.

2) В описателе временной переменной содержится номер последней тетрады, в которой используется эта переменная. После генерации команд для этой тетрады временная переменная больше не нужна.

3) Во время генерирования и оптимизации тетрад для каждой временной переменной строится список всех ссылок на нее, а начальная и конечная ссылки оставляются в описателе этой временной переменной.

11. ВЫХОДНАЯ ИНФОРМАЦИЯ КОМПИЛЯТОРА

Наш компилятор практически спроектирован, и можно обдумать вопрос о том, что же мы хотим получить от компилятора, кроме желаемого кода. Или что он нам должен сообщить?

Как правило, на старте компилятор может сообщить такую информацию, как данные о компиляторе (название, версия, производитель, дата создания) и о конкретных режимах компиляции (options in effect or supressed).

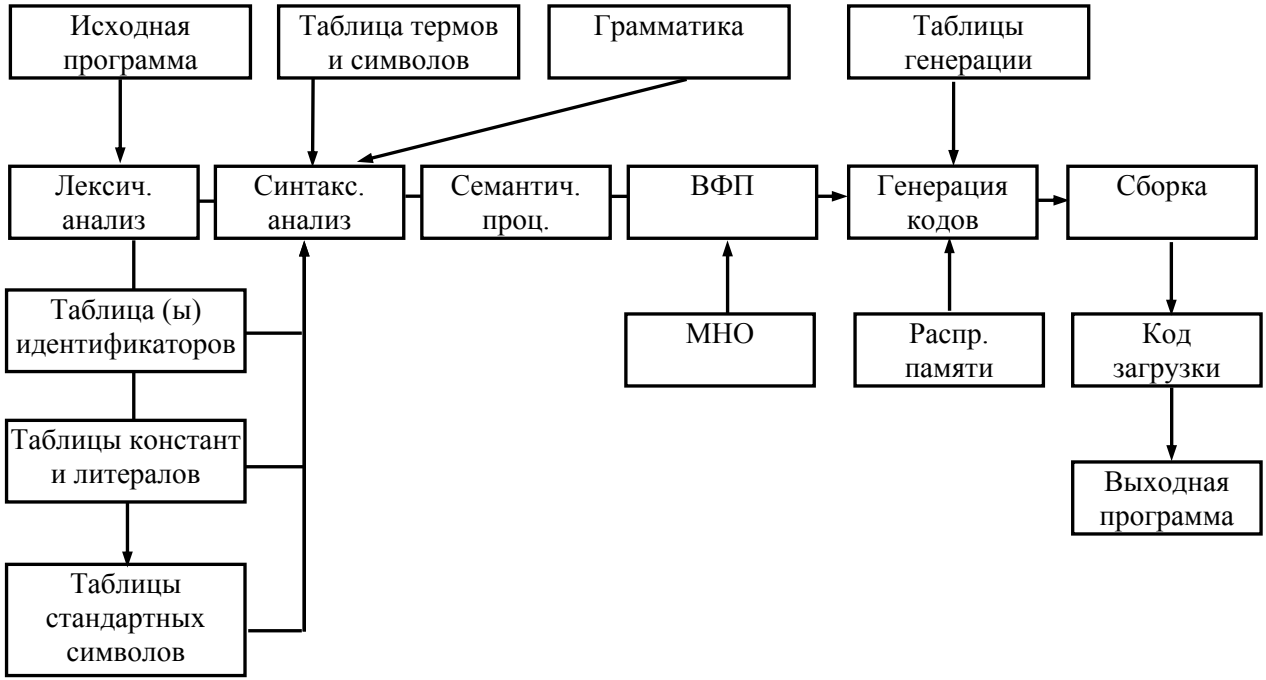
Далее – требования к листингу программы. Исполнение этих требований – тоже задача компилятора. Листинг должен четко оформить все необходимые отступы (indents), вне зависимости от того, предусмотрел их программист или нет.

При необходимости должна быть произведена нумерация строк, а также графическое изображение областей блоков программы, если такое есть в компиляторе.

В ходе процесса компиляции компилятор может встретиться с самыми различными ошибочными или неопределенными ситуациями, на которые он должен уметь адекватно реагировать. Здесь возможны такие варианты:

- Потеря управления над процессом компиляции (или зацикливание).
- Выдача некорректного кода.
- Реакция на обнаруженную ошибку – прекращение компиляции или продолжение работы.
- Определение степени серьезности тех или иных ошибок – типа Fatal Error, просто Error или Warning.
- Автокоррекция кода при компиляции с оповещением о принятых мерах и действиях.
- Наличие возможности восстановления работы компилятора – Error Recovery and Repair.

12. ОБЩАЯ СХЕМА КОМПИЛЯТОРА



13. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ №1

Приложение 1. «Программирование лексического анализатора»

1. Произвести анализ заданного программного фрагмента на языке PASCAL и выделить все типы имеющихся в нем лексем.
2. Сформировать таблицу служебных и ключевых слов и разделителей, имеющихся в заданном фрагменте.
3. Для выделенных лексем построить регулярную грамматику.
4. Для полученной грамматики построить диаграмму состояний соответствующего конечного автомата.
5. На основе автомата составить и отладить программу лексического анализатора.

13.1. Содержание отчета

Представленная к защите лабораторная работа должна быть оформлена в виде отчета, содержащего:

1. Титульный лист.
2. Краткое теоретическое обоснование.
3. Текст заданного фрагмента индивидуального задания.
4. Грамматические правила для имеющихся в сканируемом фрагменте программы лексем.
5. Диаграмму состояния соответствующего конечного автомата.
6. Текст программы сканера.
7. Таблицу распознанных лексем с указанием собственно лексемы (содержимое переменной A), типа лексемы и ее кодового представления.
8. Отпечатанный диагностический файл для исходного программного фрагмента (без лексических ошибок и с сознательно допущенными ошибками).
9. Выводы и заключения по проделанной работе.

13.2. Варианты заданий для лабораторной работы

```
    1) procedure TForm1.Button2Click(Sender:
TObject);
    var M:set of
char;ch:char;S:string;i,k:integer;
    begin
    M:=[];k:=0;
     for i:=i to ListBox1.Items.Count-1 do
      S:=S+Listbox1.Items[i];
      begin
      for i:=1 to Length(S) do
       if S[i] in M then
      M:=M+[S[i]]
                        else
     if M>5 then begin ShowMessage('Impossible')
end;end;

    2) procedure SetDisplay(R: Real);
    var  S: string[63];
    begin
      Str(R: 0: 10, S);
      if S[1] <> '-' then Sign := ' ' else
      begin
        Delete(S, 1, 1);    Sign := '-';
      end;
      if Length(S) > 15 + 1 + 10 then Error
      else
      begin
        while S[Length(S)] = '0' do Dec(S[0]);
        if S[Length(S)] = '.' then Dec(S[0]);
        Number := S;
      end; end;

    3) procedure TCalcDisplay.Draw;
    var  Color: Byte;  I: Integer;  B:
TDrawBuffer;
    begin
      Key := UpCase(Key);
     if(Status=csError) and (Key<>'C') then
Key:=' ';
      Color := GetColor(1);
      I := Size.X - Length(Number) - 2;
      MoveChar(B, ' ', Color, Size.X);
```

```
      MoveChar(B[I], Sign, Color, 1);
      MoveStr(B[I + 1], Number, Color);
      WriteBuf(0, 0, Size.X, 1, B);
    end;
    4) procedure TCalcDisplay.HandleEvent(var
Event: TEvent);
    begin
      inherited HandleEvent(Event);
      case Event.What of
        evKeyDown:
          begin
            CalcKey(Event.CharCode);
            ClearEvent(Event);
          end;
        evBroadcast:
          if Event.Command = cmCalcButton then
          begin

CalcKey(PButton(Event.InfoPtr)^.Title^[1]);
            ClearEvent(Event);
          end;
      end;
    end;

    5) constructor TCalculator.Init;
    const  KeyChar: array[0..19] of Char =
'C'#27'%'#241'789/456*123-0.=+';
    var  I: Integer;  P: PView;  R: TRect;
    begin
      R.Assign(5, 3, 29, 18); inherited Init(R,
'Calculator');
      Options := Options or ofFirstClick;
      for I := 0 to 19 do
      begin
        P:=New(PButton, Init(R,
KeyChar[I],cmCalcButton, bfNormal +
bfBroadcast));
        P^.Options := P^.Options and not
ofSelectable;
        Insert(P);
      end;  R.Assign(3, 2, 21, 3);
      Insert(New(PCalcDisplay, Init(R)));
    end;

    6) while Tmp<>nil do
```

```
        begin If  Tmp^.Num = StrToInt(S) then
          begin
           i := i+1;
           Summ := Summ + Tmp^.Mark
          end;
          tmp:=Tmp^.Next
          end;
          j:= Summ/i;
          Edit5.Text := FloatToStr(j)
        end;
```

7) procedure TForm1.Button3Click(Sender: TObject);

```
    var i:extended;R:rec;kol:integer;
    begin
     if (CEdit1.Text='') and (Kedit2.Text='')
then
    begin ShowMessage('Please enter data');exit
end
     else Reset(F);
    i:=StrTofloat(CEdit1.Text);kol:=0;
     while not EoF(F) do
       begin
          Read(F,R);      if R.Price<i then
kol:=kol+1;
       end;
       KEdit2.Text:=IntToStr(kol)
    end;
```

8) function StdEditorDialog(Dialog: Integer; Info: Pointer): Word;

```
    var  R: TRect;  T: TPoint;
    begin
      case Dialog of
      edReplace:   StdEditorDialog :=
Application^.ExecuteDialog(CreateReplaceDialog,Inf
o);
      edReplacePrompt:
          begin
    {Avoid placing the dialog on the same line as
the cursor}
           R.Assign(0, 1, 40, 8);
           R.Move((Desktop^.Size.X - R.B.X) div
2, 0);
           if TPoint(Info).Y <= T.Y then
```

```
              R.Move(0, Desktop^.Size.Y - R.B.Y -
2);
            StdEditorDialog := MessageBoxRect(R,
'Replace this occurence?',nil, mfYesNoCancel +
mfInformation);
          end;
     end;
```

9) function TIndicator.GetPalette: PPalette;

```
    const
      P: string[Length(CIndicator)] = CIndicator;
    begin
    R.A.X := (I mod 4) * 5 + 2; R.A.Y := (I div
4) * 2 + 4;
    case Operator of
    '+', '-': R := Operand * R / 100;
    '*', '/': R := R / 100;
    end;
    GetPalette := @P;
    end;
    asm     LES     DI,Buf
            MOV     CX,Count
            XOR     DX,DX
            MOV     AL,0DH
            CLD
    @@1:    JCXZ    @@2
            REPNE   SCASB
            JNE     @@2
    end;
```

10) procedure TIndicator.SetValue(ALocation:

```
TPoint; AModified: Boolean);
    begin
     case Operator of
                '+': SetDisplay(Operand + R);
                '-': SetDisplay(Operand - R);
                '*': SetDisplay(Operand * R);
                '/': if R = 0 then Error else
SetDisplay(Operand / R);
    end;
      if (Longint(Location) <>
Longint(ALocation)) or
        (Modified <> AModified) then
      begin
        Location := ALocation;
```

```
      Modified := AModified;
      DrawView;
    end;
  end;
```

```
  11) procedure TIndicator.Draw;
  var
    Color: Byte;  Frame: Char;
    L: array[0..1] of Longint;
    S: String[15];  B: TDrawBuffer;
  begin
    if State and sfDragging = 0 then
    begin
      Color := GetColor(1);
      Frame := #205;
    end else
  MoveChar(B, Frame, Color, Size.X);
    if Modified then WordRec(B[0]).Lo := 15;
    FormatStr(S, ' %d:%d ', L);
    MoveStr(B[8 - Pos(':', S)], S, Color);
    WriteBuf(0, 0, Size.X, 1, B);
  end;
```

```
  12)   procedure   TForm1.BitBtn2Click(Sender:
TObject);
  var S:string;i,n:integer;code:byte;
  begin
     S:=ListBox1.Items[0];n:=Length(S);
     for i:=1 to n do
     begin
     if S[i] in ['A'..'Z']then
          Begin
            code:=ord(S[i]);
            Inc(code,2);
            S[i]:=Chr(code)
          end
     end;
  end;
```

```
  13) constructor TEditor.Load(var S: TStream);
  begin
    inherited Load(S);
    GetPeerViewPtr(S, HScrollBar);
    GetPeerViewPtr(S, VScrollBar);
    GetPeerViewPtr(S, Indicator);
```

```
      S.Read(BufSize, SizeOf(Word));
      S.Read(CanUndo, SizeOf(Boolean));
      InitBuffer;
      if Buffer <> nil then IsValid := True else
      begin
        EditorDialog(edOutOfMemory, nil); BufSize
:= 0;
      end;
      SetBufLen(0);
    end;

    14)  procedure  TEditor.DrawLines(Y,  Count:
Integer; LinePtr: Word);
    var  Color: Word;  B: array[0..MaxLineLength
- 1] of Word;
    begin  Color := GetColor($0201);
      while Count > 0 do
      begin
        FormatLine(B, LinePtr, Delta.X + Size.X,
color);
        WriteBuf(0, Y, Size.X, 1, B[Delta.X]);
        LinePtr := NextLine(LinePtr);
        Inc(Y);
        Dec(Count);
      end;
    end;

    15) procedure TEditor.Find;
    var  FindRec: TFindDialogRec;
    begin
      with FindRec do
      begin
        Find := FindStr;
        Options := EditorFlags;
        if  EditorDialog(edFind,  @FindRec)  <>
cmCancel then
        begin
          FindStr := Find;
          EditorFlags  :=  Options  and  not
efDoReplace;
          DoSearchReplace;
        end;
      end;
```

```
    16) function TEditor.LineMove(P: Word; Count:
Integer): Word;
    var  Pos: Integer;  I: Word;
    begin
      I  :=  P;    P  :=  LineStart(P);    Pos  :=
CharPos(P, I);
      while Count <> 0 do
      begin
        I := P;    if Count < 0 then
        begin
          P := PrevLine(P);      Inc(Count);
        end else
        begin
          P := NextLine(P);      Dec(Count);
        end;
      end;
      if P <> I then P := CharPtr(P, Pos);
      LineMove := P;
    end;

    17) procedure TEditor.NewLine;
    const
      CrLf: array[1..2] of Char = #13#10;
    var  I, P: Word;
    begin
      P := LineStart(CurPtr);  I := P;
      while (I < CurPtr) and ((Buffer^[I] = ' ')
or (Buffer^[I] = #9)) do Inc(I);
      InsertText(@CrLf, 2, False);
      if AutoIndent then InsertText(@Buffer^[P],
I - P, False);
    end;

    18) procedure TEditor.SetState(AState: Word;
Enable: Boolean);
    begin
      inherited SetState(AState, Enable);
      case AState of
        sfActive:      begin
           if HScrollBar <> nil then
HScrollBar^.SetState(sfVisible, Enable);
           if VScrollBar <> nil then
VScrollBar^.SetState(sfVisible, Enable);
           if Indicator <> nil then
Indicator^.SetState(sfVisible, Enable);
            UpdateCommands;
```

```
          end;
        sfExposed:
          if Enable then Unlock;
      end;
    end;

    19)   procedure TForm1.Button2Click(Sender:
TObject);
    begin
      if  Form1.Button2.Caption='Cancel' Then
Begin Form1.Button2.Caption:='Previous';
Form1.Button1.Caption:='Next' End;
      if FilePos(F)>0 then
        begin
         Seek(F,FilePos(F)-1);
         ReadRec
        end; end;
    20) procedure TEditor.DoSearchReplace;
    var
      I: Word;  C: TPoint;
    begin
      repeat
    if not Search(FindStr, EditorFlags) then
        begin
       I := cmYes; if EditorFlags and
efPromptOnReplace <> 0 then
          if I = cmYes then
          begin
     InsertText(@ReplaceStr[1],
Length(ReplaceStr), False);
            TrackCursor(False);
          end;
        end;
    until (I = cmCancel)or (EditorFlags and
efReplaceAll = 0);
    end;

    21) procedure TEditor.DeleteRange(StartPtr,
EndPtr: Word; DelSelect: Boolean);
    begin
      if HasSelection and DelSelect then
DeleteSelect else
      begin
        CursorVisible:=(CurPos.Y >= Delta.Y) and
(CurPos.Y<Delta.Y+Size.Y);
```

```
    SetSelect(CurPtr, EndPtr, True);
DeleteSelect;    SetSelect(StartPtr, CurPtr,
False); DeleteSelect;
      end;
    end;

    22) procedure TEditor.ConvertEvent(var Event:
TEvent);
    var  ShiftState: Byte absolute $40:$17;  Key:
Word;
    begin
      if Event.What = evKeyDown then
      begin
    if KeyState <> 0 then
        Key := ScanKeyMap(KeyMap[KeyState], Key);
KeyState := 0;
        if Key <> 0 then  if Hi(Key) = $FF then
          begin
            KeyState := Lo(Key);
ClearEvent(Event);
          end else
          begin
            Event.What := evCommand;
Event.Command := Key;
          end;
      end;  end;

    23) function TEditor.CharPtr(P: Word; Target:
Integer): Word;
    var  Pos: Integer;
    begin
      Pos := 0;
    while (Pos < Target)and(P< BufLen) and
(BufChar(P) <> #13) do  begin
        if BufChar(P) = #9 then Pos := Pos + 7;
        Inc(Pos);    Inc(P);
      end;
      if Pos > Target then Dec(P);
      CharPtr := P;
    end;


    24) procedure TEditor.DoUpdate;
    begin
      if UpdateFlags <> 0 then
```

```
      begin
        SetCursor(CurPos.X - Delta.X, CurPos.Y -
Delta.Y);
        if UpdateFlags and ufView <> 0 then
DrawView
        else
          if UpdateFlags and ufLine <> 0 then
            DrawLines(CurPos.Y - Delta.Y, 1,
LineStart(CurPtr));
        if HScrollBar <> nil then
          HScrollBar^.SetParams(Delta.X, 0,
Limit.X - Size.X, Size.X div 2, 1);
        if Indicator <> nil then
Indicator^.SetValue(CurPos, Modified);
        if State and sfActive <> 0 then
UpdateCommands;
        UpdateFlags := 0;
      end;
    end;

    25) procedure TEditor.ScrollTo(X, Y:
Integer);
    begin
            CheckFirst;
            if Length(Number) = 1 then Number :=
'0' else Dec(Number[0]);
          end;
    asm
            MOV     AX,X
            CMP     AX,Y
            JLE     @@1
            MOV     AX,Y
    @@1:
      X := Max(0, Min(X, Limit.X - Size.X));
      Y := Max(0, Min(Y, Limit.Y - Size.Y));
      if (X <> Delta.X) or (Y <> Delta.Y) then
      begin
        Delta.X := X;    Delta.Y := Y;
Update(ufView);
      end; end;
```

14. ЗАДАНИЕ ЛАБОРАТОРНОЙ РАБОТЫ № 2

Приложение 2. «Программирование синтаксического анализатора»

1. Для заданного фрагмента программы на Паскале разработать грамматику для построения грамматического разбора с помощью рекурсивного спуска.
2. В качестве процедуры SCAN использовать лексический анализатор, сконструированный в лабораторной работе №1.
3. Написать и отладить процедуры рекурсивного спуска.
4. Спроектировать тест для проверки работоспособности синтаксического анализатора.

14.1. Содержание отчета

Представленная к защите лабораторная работа должна быть оформлена в виде отчета, содержащего:

1. Титульный лист.
2. Краткое теоретическое обоснование.
3. Грамматические правила для нетерминальных символов грамматики.
4. Блок-схему алгоритма с описанием примененных функций и процедур.
5. Текст программы парсера.
6. Отпечатанный диагностический файл для исходного программного фрагмента (без ошибок и с сознательно допущенными ошибками).
7. Выводы и заключения по проделанной работе.

14.2. Варианты заданий для лабораторной работы

Варианты заданий представляют собой модифицированные фрагменты текстов заданий для ЛР 1 и включают в себя все ранее распознанные лексемы, поэтому нет необходимости строить новый лексический анализатор.

```
1) for i:=i to ListBox1.Items.Count-1 do
   S:=S+Listbox1.Items[i];
```

```
    2) if Length(S) > 15 + 1 + 10 then Error
    3) if (Status = csError) and (Key <> 'C')then
Key:='';
    4) if Event.Command = cmCalcButton then
            ClearEvent(Event);
    5) for I := 0 to 19 do  P := New(PButton,
cmCalcButton);
    6) if  Tmp^.Num = StrToInt(S) then i := i+1;
    7)  if  (CEdit1.Text='')  and  (Kedit2.Text='')
then
    begin  ShowMessage('Please  enter  data');exit
end;
    8) if TPoint(Info).Y <= T.Y then
             R.Move(0, Desktop^.Size.Y - R.B.Y -
2);
    9)       P:    string[Length(CIndicator)]    =
CIndicator;
       R.A.X := (I mod 4) * 5 + 2;
    10)       if       (Longint(Location)       <>
Longint(ALocation)) or
        (Modified<>AModified)                 then
Location:=ALocation;
    11) if State and sfDragging = 0 then   Color
:= GetColor(1);
    12) if S[i] in ['A'..'Z']then
           Begin
             code:=ord(S[i]);
             Inc(code,2);
             S[i]:=Chr(code)
           End
    13)  if  Buffer  <>  nil  then  IsValid  :=  True
else
       EditorDialog(edOutOfMemory, nil);
    14) while Count > 0 do
      begin
        WriteBuf(0, Y, Size.X, 1, B[Delta.X]);
        Dec(Count);
      end;
    15) if EditorDialog(edFind, @FindRec) <>
cmCancel then  FindStr := Find;
    16) while Count <> 0 do
      begin
        P := NextLine(P);      Dec(Count);
       end;
```

```
    17) while (I < CurPtr) and Buffer^[I] = ' ')
do Inc(I);
    18) if HScrollBar <> nil then
HScrollBar^.SetState(sfVisible, Enable);
    19) if  Form1.Button2.Caption='Cancel' Then
Form1.Button2.Caption:='Previous';
    20) if EditorFlags and efPromptOnReplace <> 0
then
            TrackCursor(False);
    21) if HasSelection and DelSelect then
DeleteSelect else
       SetSelect(CurPtr, EndPtr, True)
    22) if Event.What = evKeyDown then
          begin
            KeyState := Lo(Key);
ClearEvent(Event);
          end

    23) while (P < BufLen) and (BufChar(P) <>
#13) do
      Inc(Pos);    Inc(P);
      end;
    24) if UpdateFlags <> 0 then
        SetCursor(CurPos.X - Delta.X, CurPos.Y -
Delta.Y);
    25) if Length(Number) = 1 then Number := '0'
else Dec(Number[0]);
```

15. ЗАДАНИЕ ЛАБОРАТОРНОЙ РАБОТЫ № 3

Приложение 3.

Целью работы является создание программного продукта, интегрирующего результаты лабораторных работ №1 и №2:

1) лексический и синтаксический анализы заданного исходного фрагмента программы;

2) построение необходимых таблиц идентификаторов и констант с последующей разработкой внутренней формы представления для разбираемого фрагмента в виде троек, четверок и т.д.; то есть построение семантических процедур. Рекомендуется применение четверок как вида ВФП.

В результате своей работы программа должна выдать:

1) таблицу распознанных лексем;

2) таблицу идентификаторов;

3) таблицу констант (если таковые имеются в исходном тексте);

4) массив троек (четверок и т.д.) или другую форму представления.

В отчете представить разработанные для заданного индивидуального задания четверки (тройки) и представить код генерации этой формы ВФП для заданного фрагмента.

16. РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Грис, Д. *Проектирование компиляторов для цифровых вычислительных машин*. Москва: Мир, 1975.
2. Донован, Дж. *Системное программирование*. Москва: Мир, 1975.
3. Aho, A., Sethi, R., Ullman, J. *Compilers: principles, techniques and tools.* Addison-Wesley, 1986.
4. Серебряков, В. А. *Лекции по конструированию компиляторов*. Москва: МГУ, 1993.
5. . Бек, Л. *Введение в системное проектирование*, Москва: Мир, 1988.
6. Хантер, Робин. *Основные концепции компиляторов*. Москва, Санкт-Петербург, Киев: Издательский дом «Вильямс», 2002.