**МИНИСТЕРСТВО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ**

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ
Зав. кафедрой АСУ
Профессор д-р техн. наук
_____А.М. Кориков
«____»_____2007 г.

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ МЕТОДОВ ТРАНСЛЯЦИИ

Методическое Пособие для студентов специальности 230105
«Программное обеспечение вычислительной техники
и автоматизированных систем»

Разработчик

_____В.Т.Калайда

2007

Методическое Пособие рассмотрено и рекомендовано к изданию методическим семинаром кафедры автоматизированных систем управления ТУСУР « » 2004 г.

Зав. кафедрой АСУ
проф. д-р техн. наук _____А.М. Кориков

Содержание

Введение ... 6
1. Предварительные математические сведения ... 7
1.1. Множества ... 7
1.2. Операции над множествами ... 8
1.3. Множества цепочек ... 14
1.4. Языки ... 15
1.5. Алгоритмы ... 16
1.6. Некоторые понятия теории графов ... 20
Контрольные вопросы ... 24
2. Введение в компиляцию ... 24
2.1. Задание языков программирования ... 24
2.2. Синтаксис и семантика ... 26
2.3. Процесс компиляции ... 28
2.4. Лексический анализ ... 28
2.5. Работа с таблицами ... 30
2.6. Синтаксический анализ ... 31
2.7. Генератор кода ... 32
2.8. Оптимизация кода ... 37
2.9. Исправление ошибок ... 39
2.10. Резюме ... 40
Контрольные вопросы ... 40
3. Теория языков ... 41
3.1. Способы определения языков ... 41
3.2. Грамматики ... 41
3.3. Грамматики с ограничениями на правила ... 45
3.4. Распознаватели ... 45
3.5. Регулярные множества, их распознавание и порождение ... 48
3.6. Регулярные множества и конечные автоматы ... 52
3.7. Графическое представление конечных автоматов 55
3.8. Конечные автоматы и регулярные множества ... 58
3.9. Минимизация конечных автоматов ... 58
3.10. Контекстно-свободные языки ... 62
3.10.1. Деревья выводов ... 62
3.10.2. Преобразование КС–грамматик ... 66
3.10.3. Грамматика без циклов ... 71
3.10.4. Нормальная форма Хомского ... 71
3.10.5. Нормальная формула Грейбах ... 72
3.11. Автоматы с магазинной памятью ... 75

3.11.1. Основные определения ... 75
3.11.2. Эквивалентность МП-автоматов и КС-грамматик 77
Контрольные вопросы ... 78
4. КС-грамматики и синтаксический анализ сверху вниз 79
4.1. LL(1)-грамматики ... 80
4.2. LL(1)-таблица разбора ... 86
Контрольные вопросы ... 90
5. Синтаксический анализ снизу вверх ... 91
5.1. Разбор снизу вверх ... 91
5.2. LR(1) - таблица разбора ... 93
5.3. Построение LR – таблицы разбора ... 96
5.4. Сравнение LL – и LR – методов разбора ... 99
Контрольные вопросы ... 100
6. Оптимизация кода ... 100
6.1. Оптимизация линейного участка ... 101
6.1.1. Модель линейного участка ... 101
6.1.2. Преобразование блока ... 103
6.1.3. Графическое представление блоков ... 107
6.1.4. Критерий эквивалентности блоков ... 110
6.1.5. Оптимизация блоков ... 111
6.1.6. Алгебраические преобразования ... 115
6.2. Арифметические выражения ... 120
6.2.1. Модель машины ... 120
6.2.2. Разметка дерева ... 122
6.2.3. Программы с командами STORE ... 126
6.2.4. Влияние некоторых алгебраических законов ... 126
6.3. Программы с циклами ... 138
6.3.1. Модель программы ... 138
6.3.2. Анализ потока управления ... 142
6.3.3. Примеры преобразования программ ... 146
6.3.4. Оптимизация циклов ... 149
6.4. Анализ потоков данных ... 160
6.4.1. Интервалы ... 161
6.4.2. Анализ потоков данных с помощью интервалов ... 167
6.4.3. Несводимые графы управления ... 175
7. Включение действий в синтаксис ... 179
7.1. Получение четверок ... 179
7.2. Работа с таблицей символов ... 182
Контрольные вопросы ... 184

8. Проектирование компиляторов ... 184
8.1. Число проходов ... 184
8.2. Таблицы символов ... 185
8.3. Таблица видов ... 189
Контрольные вопросы ... 191
9. Распределение памяти ... 191
9.1. Стек времени прогона ... 191
9.2. Методы вызова параметров ... 198
9.3. Обстановка выполнения процедур ... 199
9.4. «Куча» ... 200
9.5. Счетчик ссылок ... 201
9.6. Сборка мусора ... 203
Контрольные вопросы ... 208
10. Генерация кода ... 208
10.1. Генерация промежуточного кода. ... 208
10.2. Структура данных для генерации кода ... 211
10.3. Генерация кода для типичных конструкций ... 215
10.3.1. Присвоение ... 215
10.3.2. Условные зависимости ... 216
10.3.3. Описание идентификаторов ... 217
10.3.4. Циклы ... 217
10.3.5. Вход и выход из блока ... 219
10.3.6. Прикладные реализации ... 220
10.4. Проблемы, связанные с типами ... 220
10.5. Время компиляции и время прогона ... 222
Контрольные вопросы ... 223
11. Исправление и диагностика ошибок ... 224
11.1. Типы ошибок ... 224
11.2. Лексические ошибки ... 225
11.3. Ошибки в употреблении скобок ... 226
11.4. Синтаксические ошибки ... 227
11.5. Методы исправления синтаксических ошибок ... 228
11.6. Предупреждения ... 229
11.7. Сообщения о синтаксических ошибках ... 230
11.8. Контекстно-зависимые ошибки ... 231
11.9. Ошибки, связанные с употреблением типов ... 232
11.10. Ошибки, допускаемые во время прогона ... 233
11.11. Ошибки, связанные с нарушением ограничений ... 234
Контрольные вопросы ... 234
Список литературы ... 234

Введение

Настоящее пособие посвящено проблеме теоретического описания вычислительных процессов и структур. Существует достаточно большое количество вариантов организации вычислительного процесса (рис. 1.).

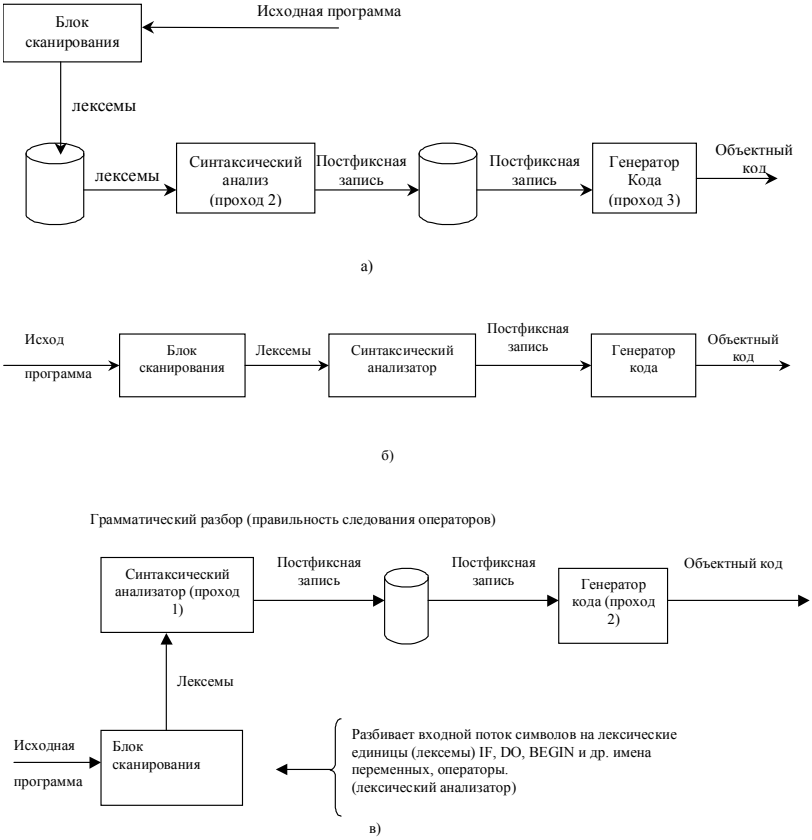


Рис 1. Схемы вариантов организации вычислительного процесса

Однако всем этим схемам присуща общая технологическая цепочка – «лексический анализ – синтаксический анализ – генерация кода – оптимизация кода». Многие элементы этой схемы в процессе развития теории программирования из интуитивных, эмпирических алгоритмов превращались в строго математически обоснованные методы, базирующиеся на теории языков, теории перевода, методах синтаксического анализа и др.

В рассматриваемом пособии используются следующие принципы:

- основное внимание уделяется теоретическим идеям, а не техническим подробностям реализации;
- широко используется принцип декомпозиции исходной задачи на составляющие, что позволяет каждую часть задачи подвергнуть оптимизации;
- изложение материала базируется на уверенности в хорошей математической подготовке слушателей.

1. Предварительные математические сведения

1.1. Множества

Будем предполагать, что существуют объекты, называемые *атомами.* Это слово обозначает первоначальное понятие, иначе говоря, термин «атом» остается неопределенным. Что называть атомом, зависит от рассматриваемой области. Часто бывает удобным считать атомами целые числа или буквы некоторого алфавита. Будем также постулировать абстрактное понятие *принадлежности.* Если a принадлежит A, то пишут $a \in A;\quad A = \{a_1, a_2, \ldots a_n\}$. Отрицание этого утверждения записывается $a \notin A$. Полагается, что, если a - атом, то ему ничто не принадлежит, т.е. $x \notin a$.

Будем также использовать некоторые примитивные объекты, называемые *множествами,* которые не являются атомами. Если A - множество, то его *элементы* – это есть объекты a (не обязательно атомы), для которых $a \in A$. Каждый элемент множества представляет собой либо атом, либо другое множество. Если A содержит конечное число элементов, то A называется *конечным* множеством.

Утверждение $\# A = n$ означает, что множество A имеет n элементов. Символ $\varnothing$ обозначает пустое множество, т.е. множество, в котором нет элементов. Заметим, что атом тоже не имеет элементов, но пустое множество не атом и атом не является пустым множеством.

Один из способов определения множества – определение с помощью *предиката*. Предикат –это утверждение, состоящее из нескольких переменных и принимающее значение 0 или 1 («*ложь*» или «*истина*»). Множество, определяемое с помощью предиката, состоит из тех элементов, для которых предикат истинен.

Говорят, что множество A *содержится* во множестве B, и пишут $A \subseteq B$, если каждый элемент из A является элементом из B. Если B содержит элемент, не принадлежащий A и $A \subseteq B$, говорят, что A, *собственно* содержится в B (рис. 1.1).

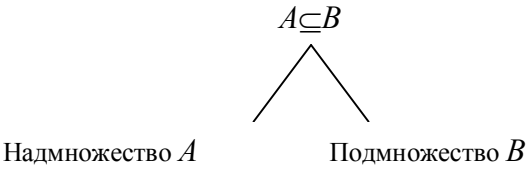


Диаграмма Венна

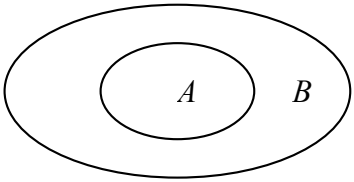


Рис. 1.1. Диаграмма Венна для включения множеств

1.2. Операции над множествами

Объединение множеств

$$A \bigcup B = \{x \mid x \in A \text{ или } x \in B\}$$ -

это множество, содержащее все элементы A и B (рис. 1.2).

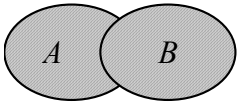


Рис. 1.2. Диаграмма для объединения множеств

Пересечение множеств

$$A \bigcap B = \{x \mid x \in A \ \text{ и } x \in B\}$$ -

это множество, состоящее из всех тех элементов, которые принадлежат обоим множествам A и B (рис. 1.3).

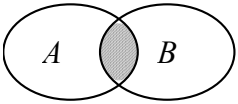


Рис. 1.3. Диаграмма Венна для пересечений множеств

Разность множеств

$A - B$ -

это множество элементов A, не принадлежащих B (рис. 1.4).

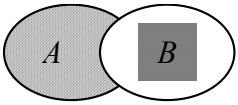


Рис. 1.4. Диаграмма Венна для разности множеств

Универсальное множество: множество всех элементов, рассматриваемых в данной ситуации, обозначается через U.

Разность U - $B = \overline{B}$ – дополнение B.

$A \cap B = \varnothing$ - A и B не пересекаются.

Определение.

Пусть I – некоторое множество, элементы которого используются как индексы, и для каждого $i \in I$ множество A_i известно. Через $\bigcup_{i \in I} A_i$ обозначим множество $\{X \mid$ существует такое $i \in I$, что $X \in A_i\}$ - это обобщенное понятие объединения.

Если I определено с помощью предиката $P(i)$, то иногда пишут $\bigcup_{P(i)} A_i$ вместо $\bigcup_{i \in I} A_i$.

Пример. $\bigcup_{i>2} A_i$ означает $A_3 \cup A_4 \cup A_5 \cdots$

Определение.

Множество всех подмножеств A обозначается через $\mathbf{P}(A)$ или 2^A т.е. $\mathbf{P}(A) = \{B \mid B \subseteq A\}$

Пример. $\mathbf{P}(A) = \{0, \{1\}, \{2\}, \{1,2\}\}$, т.е., если A конечное множество из m элементов, то $\mathrm{P}(A)$ состоит из 2^m элементов.

В общем случае элементы множества не упорядочены.

Определение.

Пусть a и b объекты. Через (a, b) обозначим упорядоченную пару объектов, взятых именно в этом порядке.

Упорядоченные пары (a, b) и (c, d) называются равными, если $a=c$ и $b=d$. Упорядоченные пары можно рассматривать как множество (a,b) - $\{a, \{a, b\}\}$.

Декартово произведение A и B

$$A \times B = \{(a,b) \mid a \in A \text{ и } b \in B\}.$$

Пример. Если $A=\{1,2\}$, $B=\{2,3,4\}$, то $A \times B=\{(1,2), (1,3), (1,4), (2,2), (2,3), (2,4)\}$

Отношения

Пусть A и B – множества. Отношением из A в B называется любое подмножество множеств $A \times B$.

Если $A=B$, то отношение задано (определено) на A.

Если R отношение из A в B и $(a, b) \in R$, то пишут aRb. Множество A называют областью определения, B - множеством значений.

Пример.

A – множество целых чисел. Отношение L представляет множество $\{(a, b) \mid a$ *меньше* $b\}$ aLb.

Определение.

Отношение $\{(b, a) \mid (a, b) \in R\}$ называют обратным к отношению R, т.е. R^{-1}.

Определение.

Пусть A – множество, R – отношение на A.

Тогда R называют:

1) *рефлексивным*, если aRa для всех пар из A;
2) *симметричным*, если aRb влечет bRa для всех a и b из A;
3) *транзитивным*, если aRb и bRc влекут aRc для a, b, c из A.

Рефлексивное, симметричное и транзитивное отношение называют отношением *эквивалентности*.

Отношение эквивалентности, определенное на A, заключается в том, что оно разбивает множество A на непересекающиеся подмножества, называемые классами эквивалентности.

Пример.

Отношение сравнения по модулю N, определенное на множестве неотрицательных чисел. a сравнимо с b по модулю N. $a \equiv b (\mathrm{mod}\, N)$, т.е. $a-b=kN$ (k - целое).

Пусть N=3, тогда множество {0, 3, 6,…, 3n,…} будет одним из классов эквивалентности, т.к. $3n=3m(mod3)$ для целых n и m. Обозначим его через [0]. Другие два:

[1]={1, 4, 7,…, 3n+1,…};

[2]={2, 5, 8,…, 3n+2,…}.

Объединение этих трех множеств дает множество неотрицательных целых чисел (рис. 1.5).

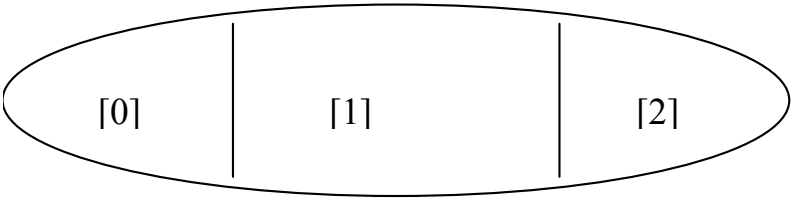


Рис. 1.5. Классы эквивалентности отношения сравнения по модулю 3

Число классов, на которые разбивается множество отношением эквивалентности, называется индексом этого отношения.

Замыкание отношений

Задача.

Для данного отношения R найти другое R', обладающее дополнительными свойствами (например, транзитивностью). Более того, желательно, чтобы R' было как можно «меньше», т.е., чтобы оно было подмножеством R.

Задача в общем случае не определена, однако для частных случаев имеет решение.

Определение.

k – степень отношения R на A $\left(R^k\right)$ определяется:

1) aR^1b тогда и только тогда, когда aRb;

2) aR^ib для $i>1$ тогда и только тогда, когда существует такое $c\in A$, что aRc и $cR^{i-1}b$.

Это пример рекурсивного определения

$\left[aR^4b;\quad aRc_1 \text{ и } c_1R^3b;\quad c_1Rc_2 \text{ и } c_2R^2b; c_2Rc_3 \text{ и } c_3R^1b\right]$.

Транзитивное замыкание отношения множества R на A (R^+) определяется так: aR^+b тогда и только тогда, когда aR^ib для некоторого $i\geq 1$.

Расшифровка понятия: aR^+b, если существует последовательность $c_1, c_2, ..., c_n$, состоящая из 0 или более элементов принадлежащих A, такая, что aRc_1, $c_1Rc_2, ..., c_{n-1}Rc_n$, c_nRb. Если n=0, то aRb.

Рефлексивное и транзитивное замыкание отношения R (R^*) на множестве A определяется следующим образом:

1) aR^*a для всех $a \in A$;
2) aR^*b, если aR^+b;
3) в R^* нет ничего другого, кроме того, что содержится в 1) и 2).

Если определить R^0, сказав, что aR^0b тогда и только тогда, когда $a=b$, то aR^*b тогда и только тогда, когда aR^ib для некоторого $i \geq 0$.

Единственное различие между R^+ и R^* состоит в том, что aR^*a истинно для всех $a \in A$, но aR^+a может быть, а может и не быть истинным.

Отношения порядка

Отношения порядка играют важную роль при изучении алгоритмов, особенно специальный вид порядков – частичный порядок.

Определение.

Частичным порядком на множестве A называют отношение R на A такое, что:

1) R – транзитивно;
2) для всех $a \in A$ утверждение aRa ложно, т.е. отношение R иррефлексивно.

Пример.

$S = \{e_1, e_2, ..., e_n\}$ - множество, состоящее из n элементов, и пусть $A=P(S)$. Положим aRb для любых a и b из A тогда и только тогда, когда $a \not\subset b$. Отношение R называется частичным порядком.

Для случая S={0, 1, 2} имеем (рис. 1.6).

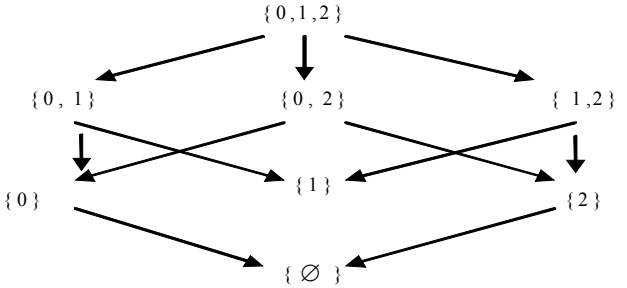


Рис. 1.6. Частичный порядок

Определение.

Рефлексивным частичным порядком называется отношение R, если

1) R – транзитивно;
2) R – рефлексивно;
3) если aRb, то $a=b$.

Последнее свойство называется *антисимметричностью.*

Каждый частичный порядок можно графически представить в виде ориентированного ациклического графа.

Линейный порядок R на множестве A – это такой частичный порядок, что, если a и $b \in A$, то либо aRb, либо bRa, либо $a=b$. Удобно это понять из следующего.

Пусть A представлено в виде последовательности $a_1, a_2, ..., a_n$, для которых $a_i R a_j$ тогда и только тогда, когда $i<j$.

Аналогично определяется рефлексивный линейный порядок.

Из традиционных систем отношение $<$ (меньше) на множестве неотрицательных целых чисел – это линейный порядок, отношение $\leq$ - рефлексивный линейный порядок.

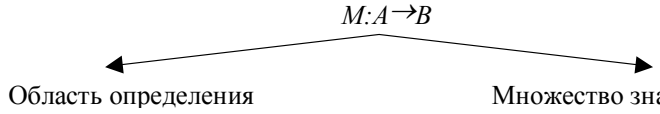
Отображение

Отображением (функцией преобразования) M множества A во множество B называют такое отношение из A в B, что, если (a, b) и (a, c) принадлежат M, то $b=c$. Если $(a, b) \in M$, то обычно пишут $M(a)=b$.

$M(a)$ определено, если существует такое $b \subset B$, что $(a, b) \in M$.

Если $M(a)$ определено для всех $a \in A$, то M всюду определено.

Если $M(a)$ определено не для всех $a \in A$, то M – частичное отображение



Если отображение $M: A \rightarrow B$ таково, что для каждого $b \in B$ существует не более одного $a \in A$ такого, что $M(a)=b$, то М называется *инъективным* (взаимно однозначным) отображением.

Если M всюду определено на A и для каждого $b \in B$ существует точно одно $a \in A$ такое, что $M(a)=b$, то M называют *биективным* отображением.

Обратное отображение обозначается M^{-1}.

Определение.

Два множества A и B называются равномощными, если существует биективное отображение A в B.

Определения:

1) множество S *конечно,* если оно равномощно множеству $\{1, 2,\ldots, n\}$ для некоторого целого n;
2) множество S *бесконечно*, если оно равномощно некоторому своему собственному подмножеству;
3) множество S *счетное*, если оно равномощно множеству положительных чисел.

1.3. Множества цепочек

Алфавитом будем называть любое множество символов (оно не обязательно конечно и даже счетно), но в наших приложениях оно конечно. Предполагается, что слово «символ» имеет достаточно ясный интуитивный смысл.

Символ – (синонимы: буква, знак) элемент алфавита.

Пример:

01011 – цепочка в бинарном алфавите {0, 1}.

Особый вид цепочки – пустая цепочка, обозначается e. Пустая цепочка не содержит символов.

Соглашение.

- Прописные буквы греческого алфавита – алфавиты.
- a, b, c и d – отдельные символы.
- t, u, v, w, x, y и z – цепочка символов.

Если цепочку из i символов обозначить $a \rightarrow a^i$, тогда $a^0=e$ – пустая цепочка.

Определение:

Цепочки в алфавите Σ определяются следующим образом:

1) e – цепочка в Σ;
2) если x цепочка в Σ и $a \in \Sigma$, то xa – цепочка в Σ;
3) y – цепочка в Σ тогда и только тогда, когда она является таковой в силу 1) и 2).

Операции над цепочками

Пусть x, y – цепочки.

- Цепочка xy – называется сцепленной (конкатенацией). Например, $x=ab$; $y=cd$; $xy=abcd$. Для любой цепочки x $xe=ex=x$.

- Обращением цепочки x $\left(x^R\right)$ называется цепочка x, записанная в обратном порядке: $x = a_1 a_2 ... a_n$, $x^R = a_n a_{n-1} ... a_1$, $e^R = e$.
- Пусть x, y, z – цепочки в некотором алфавите Σ, тогда:
  x – префикс цепочки xy;
  y – суффикс цепочки xy;
  y – называется подцепочкой цепочки xyz.

Префикс и суффикс цепочки являются ее подцепочками.

Если $x \neq y$, x – префикс (суффикс) цепочки y, то x - собственный префикс (суффикс) цепочки y.

Длина цепочки – это число символов в ней. Если $x = a_1 a_2 .... a_n$, то длина цепочки n. Длину цепочки обозначают $|x|$, например $|abc|=3$; $|e|=0$.

1.4. Языки

Языком в алфавите Σ называют множество цепочек в Σ.

Определение.

Через Σ^* обозначается множество, содержащее все цепочки в алфавите, включая e.

Например. Σ - бинарный алфавит {0,1}, тогда Σ*={e, 0, 1, 00, 01, 10, 11, 000, …,}.

Каждый язык в алфавите Σ является подмножеством Σ^*. Множество всех цепочек в Σ, за исключением e, обозначают Σ^+.

Определение. Если язык L таков, что полная цепочка в L не является собственным подмножеством (суффиксом) никакой другой цепочки в L, то L обладает префиксным (суффиксным) свойством.

Операции над языком

Так как языки являются множествами, то все операции над множествами применимы к ним. Операцию конкатенации можно применять к языкам так же, как и к цепочкам.

Определение.

Пусть L_1 язык в Σ_1., L_2 язык в Σ_2. Тогда язык $L_1 L_2$ называется конкатенацией языков L_1 и L_2 - это язык $\{xy \mid x \in L_1 \text{ и } y \in L_2\}$. Итерация языка L обозначается L^*, определяется:

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

2) $L^n = LL^{n-1}$ для $n \geq 1$;

3) $L^* = \bigcup_{n \geq 0} L^n$.

Позитивная итерация языка L обозначается L^+ - это язык $\bigcup_{n \geq 1} L^n$, т.е.

$L^* = L^+ \bigcup \{e\}$

Определение.

Пусть Σ_1 и Σ_2 - алфавиты. Гомоморфизмом называется любое отображение h: $\Sigma_1 \to \Sigma^*_2$.

Область гомоморфизма можно расширить до Σ^*_1 полагая $h(e)=e$ и $h(xa)=h(x)h(a)$ для всех $x \in \Sigma^*_1$, и $a \in \Sigma_1$.

Пример.

Если мы хотим заменить каждое вхождение в цепочку символа 0 на a, а каждое вхождение символа 1 на bb, то можно определить гомоморфизм h так: $h(0)=a$, $h(1)=bb$. Если $L = \{0^n 1^n \mid n \geq 1\}$, то $h(L) = \{a^n b^{2n} \mid n \geq 1\}$.

Определение.

Если h: $\Sigma_1 \to \Sigma^*_2$, то отношение h^{-1}: $\Sigma^*_2 \to \mathbf{P}(\Sigma^*_1)$ называется обращением гомоморфизма.

Если $y \in \Sigma^*_2$, то $h^{-1}(y)$ – это множество цепочек в алфавите Σ_1, т.е. $h^{-1}(y) = \{x \mid h(x) = y\}$. Если L – язык в алфавите Σ_2, то $h^{-1}(L)$ - язык в алфавите Σ_1 состоящий из тех же цепочек, которые h отображает в цепочки из L. Формально $h^{-1}(L) = \bigcup_{y \in L} h^{-1}(y) = \{x \mid h(x) \in L\}$

Пример.

h – гомоморфизм $h(0)=a$ и $h(1)=a$, тогда

$$h^{-1}(a) = \{0,1\};$$

$$h^{-1}(a^*) = \{0,1\}^*.$$

1.5. Алгоритмы

Алгоритм - центральное понятие в компиляции и программировании, поэтому важно его формальное определение.

Частичные алгоритмы

Неформальное определение.

Частичный алгоритм состоит из конечного числа команд, каждая из которых может выполняться механически за фиксированное время и с фиксированными затратами.

Для того чтобы быть точным, необходимо определить термин «*команда*». Кроме того, частичный алгоритм имеет любое число *входов* и *выходов*. Эти переменные тоже требуют определения.

Пример.

Алгоритм Евклида (*наибольший общий делитель*)

Вход: p и q положительные числа

Выход: g – наибольший общий делитель p и q

Метод.

Шаг 1. Найти r - остаток от деления p и q.

Шаг 2. Если r=0, положить g=q и остановиться. В противном случае положить p=q, затем q=r и перейти к шагу 1.

Данный алгоритм состоит из конечного множества команд и имеет вход и выход. Но можно ли команду выполнять механически с фиксированными затратами времени и памяти?

Строго говоря, нет. p и q могут быть очень большими и, следовательно, затраты на деление будут пропорциональны величинам p и q.

Можно заменить шаг 1 на последовательность шагов, которые вычисляют остаток от деления p на q, причем количество ресурсов, необходимых для выполнения одного такого шага, фиксировано и не зависит от p и q.

Таким образом, мы допускаем, что шаг частичного алгоритма может сам быть частичным алгоритмом.

Алгоритмы

Определение.

Частичный алгоритм останавливается на данном входе, если существует такое натуральное число t, что после выполнения t элементарных команд этого алгоритма либо не окажется ни одной команды этого алгоритма, которую нужно выполнять, либо последней выполненной командой будет «остановиться». Частичный алгоритм, который останавливается на всех входах, т.е. на всех значениях входных данных, называется всюду определенным алгоритмом либо просто алгоритмом.

Пример.

Рассмотрим предыдущий частичный алгоритм: после шага 1 выполняется шаг 2. После шага 2 либо выполняется шаг 1, либо следующий шаг невозможен, т.е. алгоритм остановлен. Можно доказать, что для каждого входа p и q алгоритм останавливается не более, чем через $2q$ шагов, и значит этот частичный алгоритм является просто алгоритмом.

Другой пример.

Шаг 1. Если $x = 0$, то перейти к шагу 1, в противном случае остановиться.

Для $x = 0$ этот частичный алгоритм никогда не остановится.

С точки зрения рассматриваемого нами предмета нас будут интересовать две проблемы:

- корректность алгоритмов;
- оценки их сложности.

При этом следует оценивать два критерия сложности:

- число выполненных элементарных механических операций как функция от величины входа (временная сложность);
- объем памяти, требующийся для хранения промежуточных результатов, возникающих в ходе вычисления, как функция от величины входа (емкостная сложность).

Пример.

Для алгоритма Евклида число шагов для $(p, q) \sim 2q$.

Объем используемой памяти 3 ячейки p, q, r.

Объем используемой памяти зависит от длины бинарного представления числа $\sim \log_2 n$, где n наибольшее из чисел p, q (объем памяти).

Рекурсивные алгоритмы

Частичный алгоритм определяет некоторое отображение множества всех входов во множество выходов. Отображение, определяемое частичным алгоритмом, называется частично рекурсивной функцией либо рекурсивной функцией. Если алгоритм всюду определен, то отображение называется общерекурсивной функцией. С помощью частичного алгоритма можно определить и язык.

Возьмем алгоритм, которому можно предъявлять произвольную цепочку x. После некоторого вычисления алгоритм выдает «да», если цепочка принадлежит языку. Если x не принадлежит языку, алгоритм останавливается и выдает «нет».

Такой алгоритм определяет L язык как множество входных цепочек, для которых он выдает «да».

Если мы определили язык с помощью всюду определенного алгоритма, то последний остановится на всех входах.

Множество, определяемое частичным алгоритмом, называется рекурсивно перечисленным.

Множество, определяемое всюду определенным алгоритмом, называется рекурсивным.

Задание алгоритмов

Мы занимались неформальным описанием алгоритмов. Можно дать строгие определения терминов, используя различные формализмы:

- машины Тьюринга;
- грамматики Хомского типа 0;
- алгоритмы Маркова;
- лямбда – исчисления;
- системы Поста;
- ТАГ – системы и др.

Языки программирования

При дальнейшем анализе мы будем пользоваться формализмом Тьюринга, вводя по мере надобности необходимые нам определения.

Проблемы

*Определение***.**

Проблема – это утверждение (предикат), истинное или ложное в зависимости от входящих в него неизвестных (переменных) определенного типа. Проблема обычно формулируется как вопрос.

Пример:

«целое число x меньше целого y».

Определение.

Частный случай проблемы – это набор допустимых значений ее неизвестных.

Отображение множества частных случаев проблемы во множество {да, нет} называется решением проблемы.

Если решение можно задать алгоритмом, то проблема называется разрешимой.

1.6. Некоторые понятия теории графов

Ориентированные графы

Определение.

Неупорядоченный ориентированный граф G – это пара (A, R):

A – множество элементов, называемых вершинами;

R – отношения на множестве A.

Пример:

$G=(A, R)$, $A=\{1, 2, 3, 4\}$, $R=\{(1, 1), (1, 2), (2, 3), (2, 4), (3, 4), (4, 1), (4, 3)\}$ (рис. 1.7).

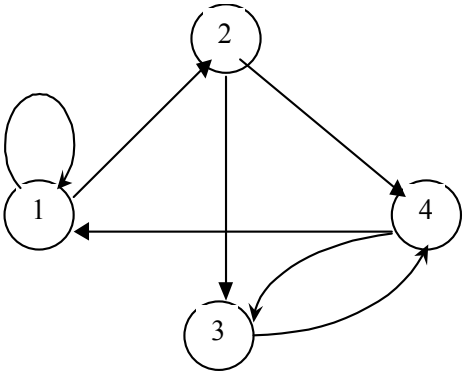


Рис. 1.7. Пример ориентированного графа

Пара (a, b) – называется дугой (или ребром) графа G.

Определение.

Пусть $G_1=(A_1, R_1)$ и $G_2=(A_2, R_2)$ - G_1 и G_2 равные (изоморфные), если существует биективное отображение $f: A_1 \rightarrow A_2$ такое, что aR_1b тогда и только тогда, когда $f(a)R_2f(b)$, т.е. в графе G_1 из вершины a в вершину b ведет дуга тогда и только тогда, когда в графе G_2 из вершины, соответствующей a, ведет дуга, соответствующая вершине b.

Части вершинам и/или дугам графа иногда приписывают некоторую информацию (разметку). Такие графы называются помеченными.

Определение.

(A, R) – граф. Разметкой графа называется пара функций f и g, где f (разметка вершины) отображает A в некоторое множество, а g (разметка дуг) отображает R в некоторое (возможно отличное от первого) множество.

Пусть $G_1=(A_1, R_1)$ и $G_2=(A_2, R_2)$ – равные помеченные графы, если существует такое биективное отображение $h: A_1,....,A_n$, что:

1) aR_1b тогда и только тогда, когда $h(a)R_2h(b)$, т.е. графы равны как непомеченные;
2) $f_1(a)=f_2(h(a))$, т.е. соответствующие вершины имеют одинаковые метки;
3) $g_1((a,b))=g_2((h(a), h(b)))$, т.е. соответствующие дуги имеют одинаковые метки.

Пример. $G_1=\{\{a, b, c\}\{(a, b), (b, c), (c, a)\}\}$ и $G_2=\{\{0, 1, 2\}\{(1, 0), (2, 1), (0, 2)\}\}$ (рис. 1.8).

Разметка графа G_1 определяется формулами:

$$f(a) = f_1(b) = x;$$

$$f_1(c) = y;$$

$$g_1((a,b)) = g_1((b,c)) = \alpha;$$

$$g_1((c,a)) = \beta.$$

Разметка графа G_2 определяется формулами

$$f_2(0) = f_2(2) = x;$$

$$f_1(1) = y;$$

$$g_2((0,2)) = g_2((2,1)) = \alpha;$$

$$g_2((1,0)) = \beta.$$

Графы равны.

Определение.

Последовательность вершин $(a_0, a_1, \ldots, a_n)$, $n\geq 1$ называется путем длины n из вершины a_0 в вершину a_n , если для каждого $1\leq i\leq n$ существует дуга, выходящая из a_{i-1} и входящая в вершину a_i.

Циклом называется путь $(a_0, a_1, \ldots, a_n)$, в котором $a_0 = a_n$.

Граф называется сильно связанным, если для двух различных вершин a и b существует путь из a в b.

Степенью по входу вершины a назовем число входящих в нее дуг, с*тепенью по выходу* – число выходящих из нее дуг.

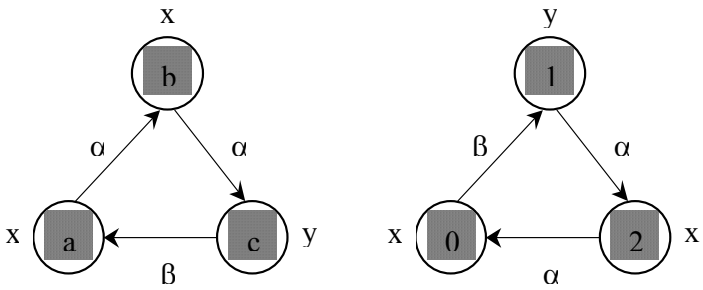


Рис. 1.8. Равные помеченные графы

Ориентированные ациклические графы

Ациклическим графом называется граф, не имеющий циклов (рис. 1.9).

Вершина, степень, по входу которой 0, называется *базовой*.

Вершина, степень, по выходу которой 0, называется *листом* (или концевой вершиной).

Если (*a, b*) – дуги ациклического графа, то *a* называется прямым предком *b*, а *b* – прямым потомком *a*.

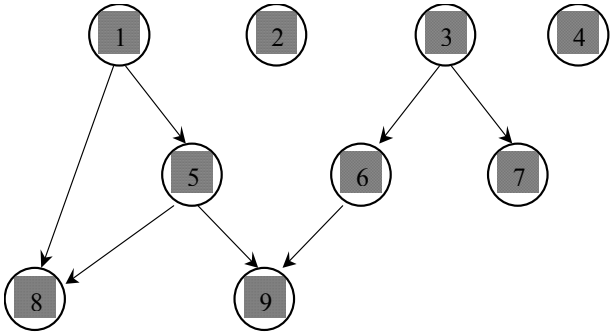


Рис. 1.9. Пример ациклического графа

Деревья

Определение.

Деревом *T* называется ориентированный граф *G*=(*A*, *R*) со специальной вершиной *r*∈*A*, называемой корнем, у которого:

1) степень по входу *r* равна 0;
2) степень по входу всех остальных вершин дерева *T* равна 1;

3) каждая вершина достижима из *r*.

Определение.

Поддеревом дерева $T=(A, R)$ называется любое дерево $T'=(A', R')$, у которого (рис. 1.10):

1) A' не пусто и содержится в A;
2) $R' = (A' \times A') \cap R$;
3) ни одна вершина $A - A'$ не является потомком вершины A'.

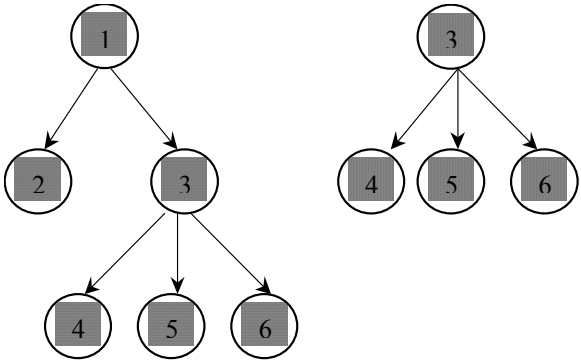


Рис. 1.10. Примеры дерева

Упорядоченные графы

Упорядоченным графом называется пара (A, R), где A - множество вершин, а R – множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид $((a, b_1), (a, b_2),\ldots,(a,b_n))$ (рис. 1.11).

Этот элемент показывает, что из вершины a выходит n дуг, причем первой из них считается дуга, приходящая в b_1 , второй - в b_2 и т.д.

Разметкой упорядоченного графа $G=(A, R)$ назовем такую пару функций f и g, что:

1) $f:A \to S$ для некоторого множества S (f помечает вершины);
2) g отображает R в последовательность символов из некоторого множества T так, что образом списка $((a, b))$. $((a, b))$ является последовательность из n символов (помеченные дуги).

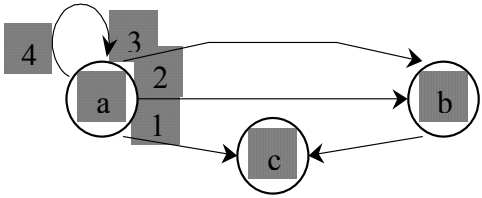


Рис. 1.11. Упорядоченный граф

Контрольные вопросы

1. Операции над множествами.
2. Отношения.
3. Замыкание отношений.
4. Отношения порядка.
5. Отображения.
6. Множества цепочек.
7. Операции над цепочками.
8. Языки.
9. Операции над языками.
10. Итерация языка.
11. Гомоморфизм.
12. Алгоритмы.
13. Частичные алгоритмы.
14. Полные алгоритмы.
15. Рекурсивные алгоритмы.
16. Задание алгоритмов.
17. Ориентированные графы.
18. Ориентированные ациклические графы.
19. Деревья. Упорядоченные графы.

2. Введение в компиляцию

2.1. Задание языков программирования

Операции машинного языка вычислительной машины значительно более примитивные, по сравнению со сложными функциями, встречающимися в математике, технике и других областях. Хотя любую функцию, которую можно задать алгоритмом, можно реализовать в виде последовательности чрезвычайно простых команд машинного языка, в большинстве приложений предпочтительнее использовать

язык высокого уровня, элементарные команды которого приближаются к типу операций, встречающихся в приложениях. Например, если выполняются матричные операции, то для выражения того обстоятельства, матрица **A** получается перемножением матриц **B** и **C**, удобнее написать команду вида

A=B*C,

чем длинную последовательность операций машинного языка.

Языки программирования могут существенно облегчить, упростить алгоритмическую запись, однако они порождают ряд новых существенных проблем, одна из них - необходимость трансляции языка программирования на машинный язык.

Другая проблема - проблема задания самого языка. Задавая язык программирования, как минимум необходимо определить:

1) множество символов, которые можно использовать для написания правильных программ;
2) множество правильных программ;
3) «смысл» правильной программы.

Первая проблема решается довольно легко. Определить множество правильных программ – это искусство.

Пример. Для многих языков программирования конструкция

L: GOTO *L*

правильная с точки зрения языка.

Самая сложная – третья проблема. Для решения третьей проблемы было предпринято несколько подходов. Один из методов заключается в определении отображения, связывающего с каждой правильной программой предложение в языке, смысл которого мы понимаем. Тогда можно определить смысл программы, записанной на любом языке программирования, в терминах *эквивалентной «программы»* в функциональном исчислении. (Под *эквивалентной программой* понимается программа, выполняющая те же самые функции).

Другой способ придать смысл программам заключается в определении идеализированной машины. Тогда смысл программы выражается в тех действиях, к которым она побуждает эту машину после того, как та начинает работу в некоторой предопределенной начальной конфигурации. В этой схеме интерпретатором данного языка становится абстрактная машина.

Третий подход – вообще игнорировать вопросы о «смысле», оставив его на совести разработчика программы. Этот подход и применяется при построении компиляторов.

Т.е. для нас «смысл» исходной программы состоит просто в выходе компилятора, когда он применяется к этой программе.

Мы будем исходить из предположения, что компилятор задан как множество пар (*x, y*),

где *x* – программа на походном языке,

y – программа в том языке, на который нужно перевести *x*.

Предполагается, что мы заранее знаем это множество, и наша главная забота – построить эффективное устройство, которое по данному входу *x* выдает выход *y*. Мы будем называть это множество пар (*x, y*) переводом. Если *x* – цепочка в алфавите Σ, а *y* – цепочка в алфавите Δ, то перевод - это просто отображение множества $\Sigma^* \rightarrow \Delta^*$.

2.2. Синтаксис и семантика

Перевод обычно рассматривают как композицию двух более простых отображений. Первое из них, называемое синтаксическим отображением, связывает с каждым выходом (программа на исходном языке) некоторую структуру, которая служит аргументом второго отображения, называемого семантическим.

Почти всегда структурой любой программы является помеченное дерево. Поэтому сущность алгоритмов перевода обычно сводится к построению подходящих деревьев для входных программ

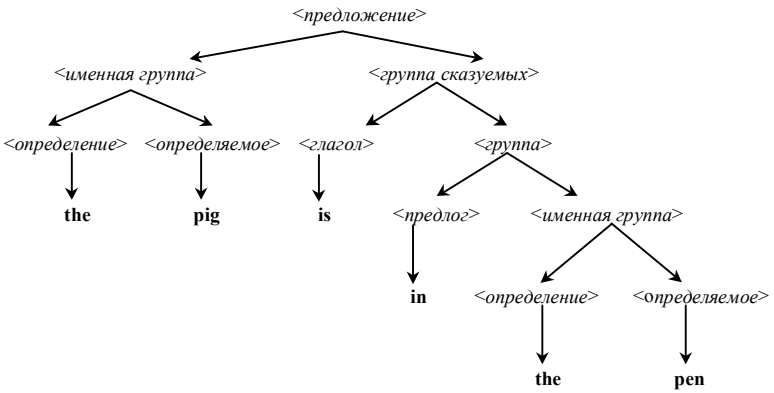


Рис. 2.1. Древовидная структура английского предложения

В качестве примера, как для цепочек строятся эти деревья, рассмотрим разбиение английского предложения на синтаксические категории (рис. 2.1).

The pig is in the pen.

Неконцевые вершины этого дерева помечены синтаксическими категориями, а концевые (листья), помечены концевыми, или терминальными, символами, в данном случае – английскими словами.

Аналогично можно программу, написанную на языке программирования, расчленить на синтаксические компоненты в соответствии с синтаксическими правилами, управляющими этим языком (рис. 2.2).

Пример.

Цепочка *a+b*c*.

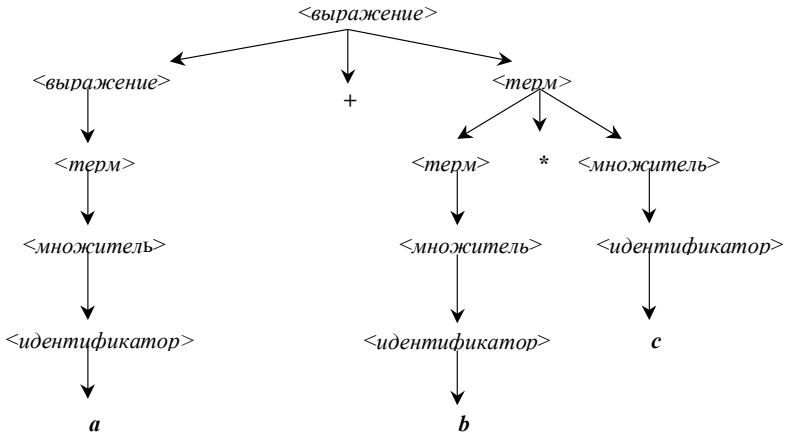


Рис. 2.2. Дерево арифметического выражения

Процесс нахождения синтаксической структуры данного предложения называется синтаксическим анализом, или синтаксическим разбором.

Синтаксический разбор позволяет понять взаимоотношения между различными частями предложения. Термином «синтаксис» языка будем называть отношения, связывающие с каждым предложением языка некоторую синтаксическую структуру, тогда правильное предложение языка можно определить как цепочку символов, синтаксическая структура которой соответствует категории «предложение».

Естественно, нам нужно более строгое определение синтаксиса. Что и будет сделано позднее.

Вторая часть перевода – семантическое отображение, оно отображает структурированный вход в выход, который обычно является программой на машинном языке.

Термином «семантика языка» будем называть отображение, связывающее с синтаксической структурой каждой входной цепочки цепоч-

ку в некотором языке, рассматриваемую как «смысл» первоначальной цепочки.

Строгой теории синтаксиса и семантики пока еще нет, однако для простых случаев – языков программирования - есть два понятия, которые можно используются для разборки части необходимого описания.

Первое из них – понятие *контекстно – свободной* (КС) грамматики. В виде контекстно – свободной грамматики можно формализовать большую часть правил, предназначенных для описания синтаксической структуры.

Второе понятие – схема синтаксически управляемого перевода, с помощью которого можно задавать отображение одного языка в другой.

Оба этих понятия – цель дальнейшего изучения.

2.3. Процесс компиляции

Практически для всех компиляторов есть некоторые общие процессы, попробуем их выделить.

Исходная программа, написанная на некотором языке, есть цепочка знаков. Компилятор превращает эту цепочку знаков в цепочку битов – объектный код. В этом процессе превращения можно выделить следующие подпроцессы:

1) лексический анализ;
2) работа с таблицами;
3) синтаксический анализ или разбор;
4) генерация кода или трансляция в промежуточный код (например, Ассемблер);
5) оптимизация кода;
6) генерация объектного кода.

В конкретных трансляторах состав и порядок этих процессов может отличаться.

Кроме того, транслятор должен быть построен так, что никакая цепочка не может нарушить его работоспособности, т.е. он должен реагировать на любые из них («защита от дурака»).

Кратко рассмотрим каждый из этих процессов.

2.4. Лексический анализ

Входом компилятора, а, следовательно, и лексического анализатора, служит цепочка символов некоторого алфавита.

Работа лексического анализатора состоит в том, чтобы сгруппировать отдельные терминальные символы в единые синтаксические объекты – лексемы. Какие объекты считать лексемами, зависит от входного языка программирования.

Лексема – цепочка терминальных символов, с которой мы связываем лексическую структуру, состоящую из пары вида (тип лексемы, некоторые данные). Первой компонентой пары является синтаксическая категория, такая как «константа» или «идентификатор», а второй указатель: в ней указывается адрес ячейки, хранящей информацию о конкретной лексеме. Для данного языка число типов лексем считается конечным.

Обычно пару (тип лексемы, указатель) называют лексемой.

Таким образом, лексический анализатор – это транслятор, входом которого служит цепочка символов, представляющая программу, а выходом – последовательность лексем.

Этот выход образует вход синтаксического анализатора.

Пример.

Оператор Фортрана

COST=(PRICE+TAX)*0.98.

Лексический анализ:

- COST, PRICE и TAX – лексемы типа <идентификатор>;
- 0.98 – лексема типа <константа>;
- =, +, * - сами являются лексемами.

Пусть все константы и идентификаторы можно отображать в лексемы типа <идентификатор>. Предполагаем, что вторая компонента лексемы представляет собой указатель элемента таблицы, содержащей фактическое имя идентификатора вместе с другими данными об этом конкретном идентификаторе.

Первая компонента используется синтаксическим анализатором для разбора.

Вторая компонента используется на этапе генерации кода для изготовления объектного модуля.

Таким образом, выходом лексического анализатора будет последовательность лексем

$$<ИД_1>=(<ИД_2>+<ИД_3>)*<ИД_4>.$$

Вторая часть компоненты лексемы (указатель) – показана в виде индексов. Символы = + * трактуются как лексемы, тип которых представляется ими самими. Они не имеют связанных с ними данных и, следовательно, не имеют указателей.

Лексический анализ легко проводить, если лексемы, состоящие более чем из одного знака, изолированы с помощью знаков, которые сами являются лексемами (*, +, =).

Однако, в общем случае лексический анализ выполнить не так легко.

Рассмотрим два правильных предложения Фортрана:

1) DO 10 I=1.15;
2) DO 10 I=1,15.

В операторе 1) цепочка DO 10 I – переменная, а цепочка 1.15 – константа.

В операторе 2) DO – ключевое слово, 10 – константа, I – переменная, 1 и 15 константы, т.е. операция «найти очередную лексему» закончится лишь тогда, когда анализатор дойдет до DO или DO 10 I. Таким образом лексических анализатор должен заглядывать вперед за интересующую его в данный момент лексему.

Другие языки, например PL/1, вообще требуют заглядывать при лексическом анализе вперед сколь угодно далеко.

Однако, существует другой подход к лексическому анализу, менее удобный, но позволяющий избежать проблемы произвольного заглядывания вперед.

Существует два крайних подхода к лексическому анализу.

- Лексический анализатор работает *прямо*, если для данного входного текста (цепочки) и положения указателя в этом тексте анализатор определяет лексему, расположенную непосредственно справа от указанного места и сдвигает указатель вправо от части текста образующего лексему.
- Лексический анализатор работает *не прямо*, если для данного текста, положения указателя в этом тексте и типа лексемы он определяет, образуют ли знаки, расположенные непосредственно справа от указателя, лексему этого типа. Если да, то указатель передвигается вправо от части текста, образующей эту лексему.

2.5. Работа с таблицами

После того, как в результате лексического анализа лексемы распознаны, информация о некоторых из них собирается и записывается в одной или нескольких таблицах. Какой характер этой информации зависит от языка программирования.

Для нашего примера на Фортране COST, PRICE и TAX – переменные с плавающей точкой.

Рассмотрим вариант такой таблицы (ее называют таблицей имен, таблицей идентификаторов или таблицей символов). В ней перечислены все идентификаторы вместе с относящейся к ним информацией (табл. 2.1).

COST = (PRICE+TAX)*0.98

Таблица 2.1 - Таблица имен

Номер элемента	Идентификатор	Информация
1	COST	Переменная с плавающей точкой
2	PRICE	Переменная с плавающей точкой
3	TAX	Переменная с плавающей точкой
4	0.98	Константа с плавающей точкой

Если позднее во входной цепочке попадается идентификатор, надо справиться в этой таблице, не появлялся ли он ранее. Если да, то лексема, соответствующая новому вхождению этого идентификатора, будет той же, что и у предыдущего вхождения.

Таким образом, таблица должна обеспечивать:

- быстрое добавление новых идентификаторов и новых сведений о них;
- быстрый поиск информации, относящейся к данному идентификатору.

Обычно применяют метод хранения данных с помощью таблиц расстановки.

Более подробно будем обсуждать этот метод далее.

2.6. Синтаксический анализ

Выходом лексического анализатора является цепочка лексем. Эта цепочка образует вход синтаксического анализатора, исследующего только первые компоненты лексемы – их типы. Информация о второй компоненте используется на более позднем этапе процесса компиляции – генерации кода.

Синтаксический анализ – разбор, в котором исследуется цепочка лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным в синтаксисе языка. Синтаксическую структуру данной цепочки важно знать также при генерации кода.

*Например. A+B*C* должна отражать тот факт, что сначала перемножается *B*C*, а потом результат складывается с *A*. При любом другом порядке операций нужное вычисление не получится.

По совокупности синтаксических правил обычно строится синтаксический анализатор, который будет проверять, имеет ли исходная программа синтаксическую структуру, определяемую этими правилами. (Далее мы рассмотрим несколько методов разбора и алгоритмов построения синтаксического анализатора).

Выходом анализатора служит дерево, которое представляет синтаксическую структуру, присущую исходной программе.

Пример.

$$<ИД_1>=(<ИД_2>+<ИД_3>)*<ИД_4>$$

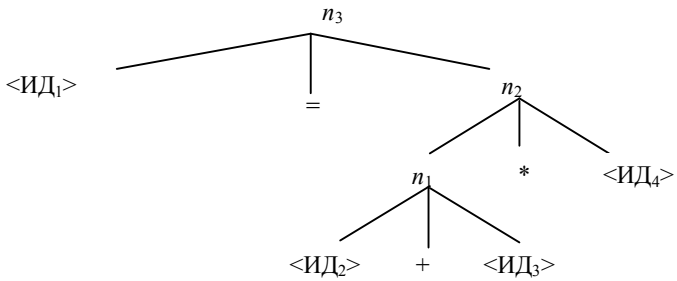
По этой цепочке необходимо выполнить

(1) $<ИД_3>$ прибавить к $<ИД_2>$

(2) результат (1) умножить $<ИД_4>$

(3) результат (2) поместить в ячейку, резервированную для $<ИД_1>$

Этой последовательности соответствует дерево:



Т.е. мы имеем последовательность шагов в виде помеченного дерева.

Внутренние вершины представляют те действия, которые можно выполнять. Прямые потомки каждой вершины либо представляют аргументы, к которым нужно применять действие (если соответствующая вершина помечена идентификатором или является внутренней), либо помогают определить, каким должно быть это действие, в частности знак +, *, =. Скобки отсутствуют, т.к. они только определяют порядок действий.

2.7. Генератор кода

Дерево, построенное синтаксическим анализатором, используется для того, чтобы получить перевод входной программы. Этот перевод может быть программой в машинном коде, но чаще всего он бывает программой на промежуточном языке, таком как ассемблер или трех-

адресный код (из операторов, содержащих не более 3 идентификаторов).

Если требуется, чтобы компилятор произвел существенную оптимизацию кода, то предпочтительно использовать трехадресный код, т.к. он не использует промежуточные регистры, привязанные к конкретному типу машин.

В качестве примера рассмотрим машину с одним регистром и команды языка типа «ассемблер» (табл. 2.2).

Запись $C(m) \rightarrow$ сумматор – означает, что содержимое ячейки памяти m надо поместить в сумматор. Запись $=m$ означает численное значение m.

Таблица 2.2. - Команды «типа запись ассемблер»

Команда	Действие
LOAD m	$C(m)\rightarrow$сумматор
ADD m	C(сумматор)+$C(m)\rightarrow$сумматор
MPY m	C(сумматор)*$C(m)\rightarrow$сумматор
STORE m	C(сумматор)$\rightarrow m$
LOAD $=m$	$m\rightarrow$сумматор
ADD $=m$	C(сумматор)+$m\rightarrow$сумматор
MPY $=m$	C(сумматор)*$m\rightarrow$сумматор

С помощью дерева, полученного синтаксическим анализатором, и информации, хранящейся в таблице имен, можно построить объектный код.

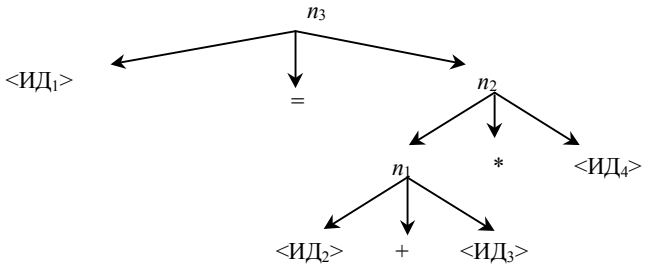
Существует несколько методов построения промежуточного кода по синтаксическому дереву. Наиболее изящный из них называется *синтаксическим управляемым переводом*. В нем с каждой вершиной n связывается цепочка $C(n)$ промежуточного кода. Код для этой вершины n строится сцеплением в фиксированном порядке кодовых цепочек, связанных с прямыми потомками вершины n, и некоторых фиксированных цепочек. Процесс перевода идет, таким образом, снизу вверх (от листьев к корню). Фиксированные цепочки и фиксированный порядок задаются используемым алгоритмом перевода.

Здесь возникает важная проблема: для каждой вершины n необходимо выбрать код $C(n)$ так, чтобы код, приписываемый корню, оказывался искомым кодом всего оператора. Вообще говоря, нужна какая-то интерпретация кода $C(n)$, которой можно было бы единообразно пользоваться во всех ситуациях, где встретится вершина n.

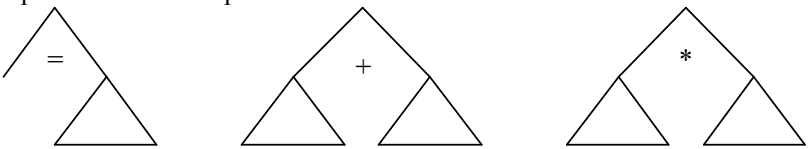
Для математических операторов присвоения нужная интерпретация получается весьма естественно. В общем случае при применении син-

таксически управляемой трансляции интерпретация должна задаваться создателем компилятора.

В качестве примера рассмотрим синтаксически управляемую трансляцию арифметических выражений. Вернемся к исходному дереву.



Допустим, что есть три типа внутренних вершин, зависящих от того, каким из знаков помечен средний потомок =, +, *. Эти три типа вершин можно изобразить:



а б в

где Δ - произвольные поддеревья (в том числе состоящие из единственной вершины).

Для любого арифметического оператора присвоения, включающего только арифметические операции + и *, можно построить дерево с одной вершиной (типа *а*) и остальными вершинами только типов *б* и *в*.

Код соответствующей вершины будет иметь следующую интерпретацию:

1) если *n* – вершина типа *а*, то *C*(*n*) будет кодом, который вычисляет значение выражения, соответствующее правому поддереву, и помещает его в ячейку, зарезервированную для идентификатора, которым помечен левый поток;
2) если *n* – вершина типа *б* или *в*, то цепочка LOAD *C*(*n*) будет кодом, засылающим в сумматор значение выражения, соответствующего поддереву, над которым доминирует вершина *n*.

Так, для нашего дерева код LOAD $C(n_1)$ засылает в сумматор значение выражения <ИД$_2$>+<ИД$_3$>, код LOAD $C(n_2)$ засылает в сумматор значение выражения (<ИД$_2$>+<ИД$_3$>)*<ИД$_4$>, а код $C(n_3)$ засылает в сумматор значение последнего выражения и помещает его в ячейку, предназначенную для <ИД$_1$>.

Теперь надо показать, как код $C(n)$ строится из кодов потомков вершины n. В дальнейшем мы будем предполагать, что операторы языка ассемблера записываются в виде одной цепочки и отделяются друг от друга точкой с запятой или началом новой строки. Кроме того, мы будем предполагать, что каждой вершине n дерева приписывается число $l(n)$, называемое уровнем, которое означает максимальную длину пути от этой вершины до листа, т.е. $l(n)=0$, если n – лист, а если n имеет потомков $n_1, n_2, ..., n_k$, то $l(n) = \max\limits_{1\le i\le k}(l(n_i)+1)$.

Уровни $l(n)$ можно вычислить снизу вверх одновременно с вычислением кодов $C(n)$ (рис. 2.3).

Уровни записываются, для того чтобы контролировать использование временных ячеек памяти. Две нужные нам величины нельзя заслать в одну и ту же ячейку памяти.

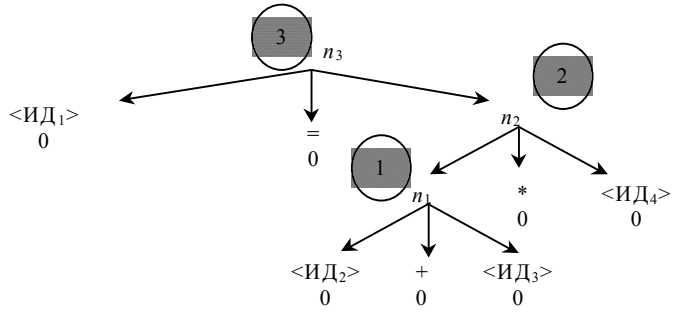


Рис. 2.3. Дерево с уровнями

Теперь определим синтаксически управляемый алгоритм генерации кода, предназначенный для вычисления кода $C(n)$ всех вершин дерева, состоящих из листьев корня типа *а* и внутренних вершин типа *б* и *в*.

Алгоритм.

Вход.

Помеченное упорядоченное дерево, представляющее собой оператор присвоения, включающий только арифметические операции * и +. Предполагается, что уровни всех вершин уже вычислены.

Выход.

Код в ячейке ассемблера, вычисляющий этот оператор присвоения.

Метод.

Делать шаги 1) и 2) для всех вершин уровня 0, затем для вершин уровня 1 и т.д., пока не будут отработаны все вершины.

1) Пусть n – лист с меткой <ИД*i*>.
   (i) Допустим, что элемент i таблицы идентификаторов является переменной. Тогда $C(n)$ – имя этой переменной.
   (ii) Допустим, что элемент j таблицы идентификаторов является константой k, тогда $C(n)$ – цепочка $=k$.
2) Если n – лист с меткой =, +, *, то $C(n)$ – пустая цепочка.
3) Если n – вершина типа *а* и ее прямые потомки – это вершины n_1 n_2 n_3, то $C(n)$ – цепочка LOAD $C(n_3)$; STORE $C(n_1)$.
4) Если n – вершина типа *б* и ее прямые потомки – это вершины n_1 n_2 n_3 ,то $C(n)$ – цепочка $C(n_3)$; STORE \$$l(n)$; LOAD $C(n_1)$; ADD \$$l(n)$. Эта последовательность занимает временную ячейку, именем которой служит \$ вместе со следующим за ним уровнем вершины n. Непосредственно видно, что, если перед этой последовательностью поставить LOAD, то значение, которое она поместит в сумматор, будет суммой значений выражением поддеревьев, над которыми доминируют вершины n_1 и n_3. Выбор имен временных ячеек гарантирует, что два нужных значения одновременно не появятся в одной ячейке.
5) Если n – вершина типа *в*, а все остальное как и в 4), то $C(n)$ – цепочка $C(n_3)$; STORE \$$l(n)$; LOAD $C(n_1)$; MPY \$$l(n)$.

Применим этот алгоритм к нашему примеру (рис. 2.4).

Таким образом, в корне мы получили ассемблеровскую программу, эквивалентную фортрановской:

COST=(PRICE+TAX)*0.98.

Естественно, эта ассемблеровская программа далека от оптимальной, но это можно исправить на этапе оптимизации.

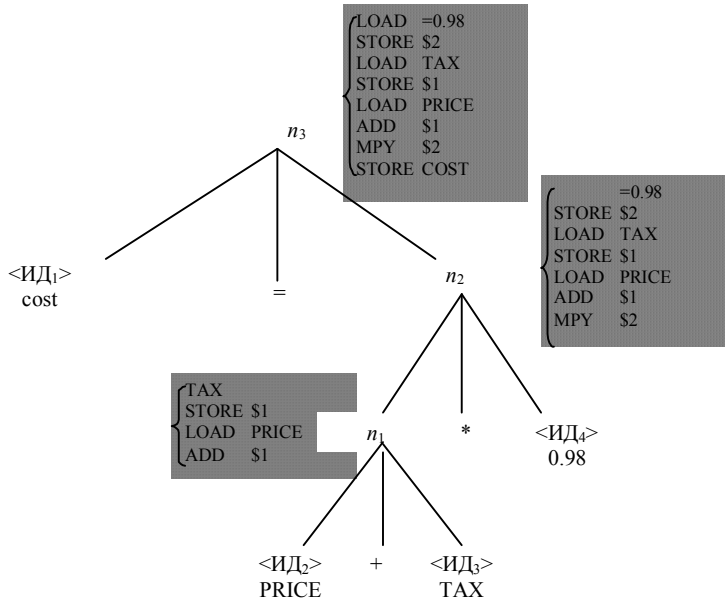


Рис. 2.4. Дерево с генерированными кодами

2.8. Оптимизация кода

Во многих случаях желательно иметь компилятор, который бы создавал эффективно работающие объектные программы. Термин *оптимизация кода* применяется к попыткам сделать объектные программы более «эффективными», т.е. быстрее работающими или более компактными.

Для оптимизации кода существует широкий спектр возможностей. На одном конце находятся истинно оптимизирующие алгоритмы. В этом случае компилятор пытается составить представление о функции, определяемой алгоритмом, программа которого записана на исходном языке. Если он «догадается», что это за функция, то может попытаться заменить прежний алгоритм новым, более эффективным алгоритмом, вычисляющий ту же функцию, и уже для этого алгоритма генерировать машинный код.

К сожалению, оптимизация этого типа чрезвычайно трудна, т.к. нет алгоритмического способа нахождения самой короткой или самой быстрой программы, эквивалентной данной.

Поэтому в общем случае термин «оптимизация» совершенно неправильный – на практике мы должны довольствоваться *улучшением кода*. На разных стадиях процесса компиляции применяются различные приемы улучшения кода.

В общем случае мы должны выполнить над данной программой последовательность преобразований в надежде повысить ее эффективность. Эти преобразования должны, разумеется, сохранить эффект, создаваемый во внешнем мире исходной программой.

Преобразования можно проводить в различные моменты компиляции, начиная от входной программы, заканчивая фазой генерации кода.

Более подробно оптимизацией кода мы займемся далее.

Сейчас рассмотрим лишь те приемы, которые делают код более коротким.

1) Если + - коммутативная операция, то можно заменить последовательность команд LOAD α; ADD β; последовательностью LOAD β; ADD α. Требуется, однако, чтобы в других местах не было перехода к оператору ADD β.
2) Подобным же образом, если * - коммутативная операция, то можно заменить LOAD α; MPY β на LOAD β; MPY α.
3) Последовательность операторов типа STORE α; LOAD α можно удалить из программы при условии, что либо ячейка α не будет использоваться далее, либо перед использованием ячейка α будет заполнена заново.
4) Последовательность LOAD α; STORE β; можно удалить, если за ней следует другой оператор LOAD и нет перехода к оператору STORE β, а последующие вхождения β будут заменены на α вплоть до того места, где появится другой оператор STORE β.

Рассмотрим наш пример. Мы получили программу (табл. 2.3).

Таблица 2.3 - Оптимизация кода

LOAD =0.98 STORE $2 LOAD TAX STORE $1 LOAD $1 ADD PRICE MPY $2 STORE COST	LOAD =0.98 STORE $2 LOAD TAX ADD PRICE MPY $2 STORE COST	LOAD TAX ADD PRICE MPY =0,98 STORE COST
Применяем правило 1) к последовательности LOAD PRICE; ФВВ ;1 Заменяем на LOAD $1 ADD PRICE	Удаляем последовательность STORE $1; LOAD $1	К последовательности LOAD =0.98; STORE $2 Применяем правило 4) и удаляем их. В команде MPY $2 Заменяется $2 на MPY =0,98

2.9. Исправление ошибок

Предположим, что входом компилятора служит правильно построенная программа (однако, на практике очень часто это не так).

Компилятор имеет возможность обнаружить ошибки в программе по крайней мере на трех этапах компиляции:

- лексического анализа;
- синтаксического анализа;
- генерации кода.

Если встретилась ошибка, то компилятору трудно по неправильной программе решить, что имел в виду ее автор. Но в некоторых случаях легко сделать предположение о возможном исправлении программы.

Например, если *A=B*2C*, то вполне правдоподобно допустить *A=B*2*C*. В общем случае компилятор зафиксирует эту ошибку и остановится. Однако некоторые компиляторы стараются провести минимальные изменения во входной цепочке, чтобы продолжить работу.

Перечислим несколько возможных изменений.

- Замена одного знака. Если лексический анализатор выдает синтаксическое слово INTE**J**ER в неподходящем для появления иденти-

фикатора месте программы, то компилятор может догадаться, что подразумевается слово INTE**G**ER.

- Вставка одной лексемы, т.е. заменить *2C* на *2*C*.
- Устранение одной лексемы. DO 10 I=1,20,.
- Простая перестановка лексем. I INTEGER на INTEGER I.

Далее мы подробно остановимся на реализации таких компиляторов.

2.10. Резюме

На рис. 2.5. приведена принципиальная модель компилятора, которая является лишь первым приближением к реальному компилятору. В реальности фаз может быть значительно больше, т.к. компиляторы должны занимать как можно меньший объем памяти.

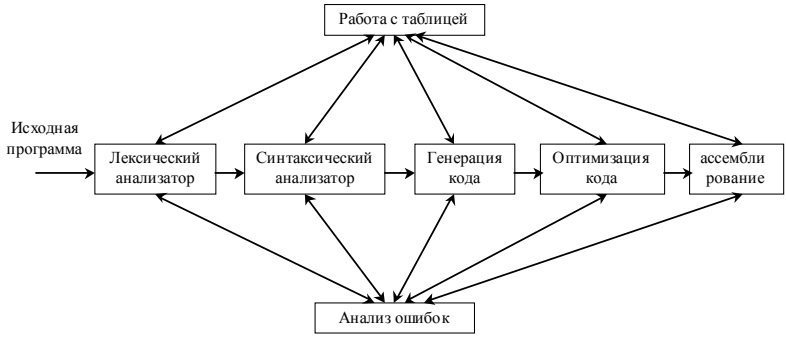


Рис 2.8. Модель компилятора

Мы будем интересоваться фундаментальными проблемами, возникающими при построении компиляторов и других устройств, предназначенных для обработки языков.

Контрольные вопросы

1. Задание языков программирования.
2. Синтаксис и семантика.
3. Процесс компиляции.
4. Лексический анализ.
5. Работа с таблицами.
6. Синтаксический анализ.
7. Генерация кода.
8. Алгоритм генерации кода.

9. Оптимизация кода.
10. Исправление ошибок.

3. Теория языков

3.1. Способы определения языков

Мы определяем язык L как множество цепочек *конечной* длины в алфавите $\sum$.

Первый вопрос - как описать язык L в том случае, когда он бесконечен. Если L состоит из конечного числа цепочек, то самый очевидный способ – составить список всех цепочек.

Однако для многих языков нельзя установить верхнюю границу длины самой длинной цепочки. Следовательно, приходится рассматривать языки, содержащие сколь угодно много цепочек. Очевидно, такие языки нельзя определить исчерпывающим перечислением входящих в них цепочек, и необходимо искать другой способ их описания. И как прежде, мы хотим, чтобы описание языков было конечным, хотя описываемый язык может быть бесконечным.

Известно несколько способов описания языков, удовлетворяющих этим требованиям. Один из способов состоит в использовании порождающей системы, называемой *грамматикой.*

Цепочки языка строятся точно определённым способом с применением правил грамматики. Одно из преимуществ определения языка с помощью грамматики состоит в том, что операции, проводимые в ходе синтаксического анализа и перевода, можно сделать проще, если воспользоваться структурой, которую грамматика приписывает цепочкам (предложениям).

Второй метод описания языка – частичный алгоритм, который для произвольной входной цепочки останавливается и отвечает «да» после конечного числа шагов, если эта цепочка принадлежит языку.

Мы будем представлять частичный алгоритм, определяющий языки, в виде схематизированного устройства, которое будем называть *распознавателем.*

3.2. Грамматики

Грамматики образуют наиболее важный класс генераторов языка. Грамматика – это математическая система, определяющая вид языка. Одновременно она является устройством, которое придаёт цепочкам

(предложениям) языка полезную структуру. Мы будем пользоваться формализмом грамматик Хомского.

В грамматике, определяющей язык L, используются два конечных непересекающихся множества символов – множество нетерминальных символов, которое обычно обозначается буквой N, и множество терминальных символов, обозначаемое Σ. Из терминальных символов образуются слова (цепочки) определяемого языка. Нетерминальные символы служат для порождения слов языка L определённым способом.

Сердцевину грамматики составляет конечное множество P правил образования, которое описывает процесс порождения цепочек языка. Правило – *это* просто пара цепочек или элемент множества, иначе говоря, $(N\cup\Sigma)^*\ N\ (N\cup\Sigma)^* \times (N\cup\Sigma)^*$. Первой компонентой правила является любая цепочка, содержащая хотя бы один нетерминал, а второй компонентой – любая цепочка.

Пример.

Например, правилом может быть пара (*AB, CDE*). Если уже установлено, что некоторая цепочка α порождается грамматикой и α содержит *AB*, т.е. левую часть этого правила, в качестве своей подцепочки, то можно образовать новую цепочку β, заменив одно вхождение *AB* в α на *CDE.*

Язык, определяемый грамматикой, - это множество цепочек, которые строятся только из терминальных символов и выводятся, начиная с одной особой цепочки, состоящей из одного выделенного символа, обычно обозначаемого S.

Соглашение. Правило (α, β) будем записывать $\alpha\rightarrow\beta$.

Определение.

Грамматикой называется четвёрка G=(N, Σ, P, S),

где

N – конечное множество нетерминальных символов или нетерминалов (иногда называемых вспомогательными символами, синтаксическими переменными или понятиями);

Σ - непересекающееся с N конечное множество терминальных символов (терминалов);

P – конечное подмножество множества $(N\cup\Sigma)^*N(N\cup\Sigma)^*\times(N\cup\Sigma)^*$, элемент (α, β) множества P называется правилом (или продукцией) и записывается $\alpha\rightarrow\beta$;

S – выделенный символ из N, называемый начальным (исходным) символом.

Примером грамматики служит четвёрка G_1 = ({A, S}, {0, 1}, P, S), где P состоит из правил

S→0A1
0A→00A1
A→*e*.

Нетерминальными символами являются A и S, а терминальными - 1 и 0.

Грамматика определяет язык рекурсивным образом. Рекурсивность проявляется в задании особого рода цепочек, называемых вводимыми цепочками грамматики *G*=(N,Σ,P,S), где

1) S - вводимая цепочка;
2) если αβγ – выводимая цепочка и β→δ содержится в P, то αδγ – тоже выводимая цепочка.

Выводимая цепочка грамматики *G,* не содержащая нетерминальных символов, называется терминальной цепочкой, порождаемой грамматикой G.

Терминология.

Пусть *G*=(N, Σ, P, S) – грамматика. Отношение $\Rightarrow_G$ на множестве $(N\cup\sum)^*$ ($\varphi\Rightarrow_G\Psi$ означает Ψ, непосредственно выводимая из φ) и практикуется: если αβγ – цепочка из $(N\cup\sum)^*$ и β→δ – правило из P, то αβγ $\Rightarrow_G$ αδγ.

Транзитивное замыкание отношения $\Rightarrow_G$ обозначим через $\Rightarrow_{G+}$ и трактуется – Ψ, выводимая из φ нетривиальным образом.

Рефлексивное и транзитивное замыкание отношения $\Rightarrow_G$ ($\Rightarrow_G$*) φ$\Rightarrow_G$* Ψ, Ψ, выводимая из φ.

Далее, если ясно, о какой грамматике идёт речь, то индекс *G* будет опускаться.

Таким образом, $L(G) = \{w \mid w\in\sum^*, S\Rightarrow^* w\}$ через $\Rightarrow^k$ будем обозначать *k*-ю степень отношения ⇒. Иначе говоря $\alpha\Rightarrow^k\beta$, если существует $\alpha_0,\alpha_1,...,\alpha_k$, состоящая из *k*+1 цепочек, для которых $\alpha=\alpha_0,...,\alpha_{i-1},\alpha_i$ при $1\le i\le\kappa$ и α_k=β. Эта последовательность цепочек называется выводом длины *k* цепочки β из цепочки α в грамматике *G*.

Отметим, что $\alpha\Rightarrow^*\beta$ тогда и только тогда, когда $\alpha\Rightarrow^i\beta$ для некоторого $i\ge0$, и $\alpha\Rightarrow^+\beta$ тогда и только тогда, когда $\alpha\Rightarrow^i\beta$ для некоторого $i\ge1$.

Пример.

Рассмотрим грамматику G_1 из ранее приведённого примера

S⇒0A1⇒00A11⇒0011.

На первом шаге S заменяется на A01 в соответствии с правилом S→A01.

На втором шаге 0A заменяется на 00A1.

На третьем шаге A заменяется на *e*.

Можно сказать, что $S \Rightarrow^{3} 0011$

$S \Rightarrow^{+} 0011$

$S \Rightarrow^{*} 0011$,

и что 0011 принадлежит языку $L(G_1)$.

Соглашение: $\alpha \to \beta_1$

$\alpha \to \beta_2$

…….

$\alpha \to \beta_n$

обозначим $\alpha \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$.

Кроме того, примем ещё следующие соглашения относительно символов и цепочек, связанных с грамматикой:

(1) *a, b ,c, d* и цифры 0,1,2,..,9 обозначают терминальные символы;

(2) A, B, C, D, S обозначают нетерминалы, S – начальный символ;

(3) U, V,...,Z обозначают либо нетерминалы, либо терминалы;

(4) α, β,.. обозначают цепочки, которые могут содержать как терминалы, так и нетерминалы;

(5) *u, v,...,z* обозначают цепочки, состоящие только из терминалов.

Пример.

Пусть G_0 = ({E, T, F},{*a*, +, * (,)}, P, E), где P состоит из правил:

E→E+T | T

T→T*F | F

F→(E) | *a.*

Пример вывода в этой грамматике:

E⇒E+T

⇒T+T

⇒F+T

⇒*a*+T

⇒*a*+T*F

⇒*a*+F*F

⇒*a*+*a**F

⇒*a*+*a***a,*

т.е. язык G_0 представляет собой множество арифметических выражений, построенных из символов *a*, +, *, (и).

3.3. Грамматики с ограничениями на правила

Грамматики можно классифицировать по виду их правил: пусть $G=(N, \Sigma, P, S)$ – грамматика.

Определение. Грамматика G называется:

1) *праволинейной,* если каждое правило из P имеет вид $A \to xB$ или $A \to x$, где $A, B \in N$;
2) *контекстно-свободной* (или *бесконтекстной*), если каждое правило из P имеет вид $A \to \alpha$, где $A \in N$, $\alpha \in (N \cup \Sigma)^*$;
3) *контекстно-зависимой* (или *неукорачивающей*), если каждое правило из P имеет вид $\alpha \to \beta$. $|\alpha| \le |\beta|$.

Грамматика, не удовлетворяющая ни одному из заданных ограничений, называется грамматикой общего вида (грамматика без ограничений).

Рассмотренный ранее пример – множество арифметических выражений, построенных из символов a + *, является примером контекстно–свободной грамматики.

Заметим, что согласно введённым определениям, каждая праволинейная грамматика – контекстно-свободная грамматика. Контекстно-зависимая грамматика запрещает правило $A \to e$ (e – правило).

Соглашение. Если язык L порождается грамматикой типа x, то L называется языком типа x. Это соглашение относится ко всем «типам x».

Определённые нами выше типы грамматик и языков называют *иерархией Хомского*.

3.4. Распознаватели

Второй распространённый метод, обеспечивающий задание языка конечными средствами, состоит в использовании распознавателей. В сущности, распознаватель – это схематизированный алгоритм, определяющий некоторое множество.

Распознаватель состоит из трёх частей (рис 3.1) - *входной ленты*, *управляющего устройства* с конечной памятью и *вспомогательной* или *рабочей, памяти*.

Входную ленту можно рассматривать как линейную последовательность клеток, каждая ячейка которой содержит один символ из некоторого конечного входного алфавита. Самую левую и самую правую ячейки обычно занимают (хотя и необязательно) маркеры.

Входная головка в каждый данный момент читает (обозревает) одну входную ячейку. За один шаг работы распознавателя входная го-

ловка может двигаться на одну ячейку влево, оставаться неподвижной, либо двигаться на одну ячейку вправо.

Памятью распознавателя может быть любого типа хранилище информации. Предполагается, что алфавит памяти конечен и хранящаяся в памяти информация построена только из символов этого алфавита. Предполагается также, что в любой момент времени можно конечными средствами описать содержимое и структуру памяти, хотя с течением времени память может становиться сколь угодно большой.

Поведение вспомогательной памяти для заданного класса распознавателей можно охарактеризовать с помощью двух функций: функции *доступа* и функция *преобразования памяти*.

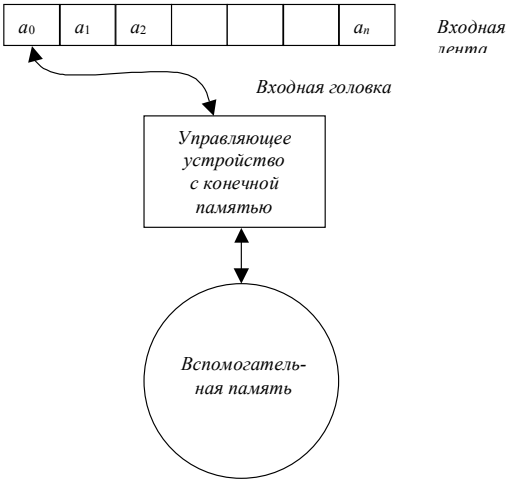


Рис. 3.1. Распознаватель

Функция доступа к памяти – это отображение множества возможных состояний или конфигураций памяти в конечное множество информационных символов.

Функция преобразования памяти – это отображения, описывающие её изменения. Вообще, именно тип памяти определяет название распознавателя (распознаватель магазинного типа).

Управляющее устройство – это программа, управляющая поведением распознавателя. Управляющее устройство представляет собой конечное множество состояний вместе с отображением, которое описывает, как меняются состояния в соответствии с текущим входным

символом (т.е. находящимся под входной головкой) и текущей информацией, извлечённой из памяти. Управляющее устройство определяет также, в каком направлении сдвинуть головку и какую информацию поместить в память.

Распознаватель работает, проделывая некоторую последовательность шагов или тактов.

В начале такта читается текущий входной символ и с помощью функций доступа исследуется память. Текущий символ и информация, извлечённая из памяти, вместе с текущим состоянием управляющего устройства определяет, каким должен быть такт. Собственно такт состоит из следующих моментов:

1) входная головка сдвигается на одну ячейку влево, вправо или остаётся в исходном положении;
2) в памяти помещается некоторая информация;
3) изменяется состояние управляющего устройства.

Поведение распознавателя обычно описывается в терминах конфигураций распознавателя. *Конфигурация* – это «мгновенный снимок» распознавателя, на котором изображены:

1) состояние управляющего устройства;
2) содержимое входной ленты вместе с положением головки;
3) содержимое памяти.

Управляющее устройство может быть *детерминированным* либо *недетерминированным*.

В детерминированном устройстве для каждой конфигурации существует не более одного возможного следующего шага.

Недетерминированное устройство – это просто удобная математическая абстракция, не реализуемая на практике.

Конфигурация называется *начальной*, если управляющее устройство находится в начальном состоянии – входная головка обозревает самый левый символ, и память имеет заранее установленное начальное содержимое.

Конфигурация называется *заключительной*, если управляющее устройство находится в одном из состояний заранее выделенного множества заключительных состояний, а входная головка обозревает правый концевой маркер.

Распознаватель допускает входную цепочку ω, если, начиная с начальной конфигурации, в которой цепочка ω записана на входной ленте, распознаватель может проделать конечную последовательность шагов, заканчивающуюся конечной конфигурацией.

Язык, определяемый распознавателем – это множество входных цепочек, которые он допускает.

Для каждого класса грамматик из иерархии Хомского существует естественный класс распознавателей:

1) язык L праволинейный тогда и только тогда, когда он определяется односторонним детерминирванным конечным автоматом;
2) язык L – контекстно-свободный тогда и только тогда, когда он определяется односторонним недетерминированным автома-том с магазинной памятью;
3) язык L контекстно-зависимый тогда и только тогда, когда он определяется двусторонним недетерминированным линейно-ограниченным автоматом;
4) язык L рекурсивно перечисляемый тогда и только тогда, когда он определяется машиной Тьюринга.

3.5. Регулярные множества, их распознавание и порождение

Рассмотрим методы задания языков программирования и класс множеств, образующий этот класс языков. Основным аппаратом задания будут регулярные множества и регулярные выражения на них.

Определение.

Пусть Σ - конечный алфавит. Регулярное множество в алфавите Σ определяется рекурсивно следующим образом:

1) $\varnothing$ – (пустое множество) – регулярное множество в алфавите Σ;
2) $\{e\}$ – регулярное множество в алфавите Σ;
3) $\{a\}$ – регулярное множество в алфавите Σ для каждого $a \in \Sigma$;
4) если P и Q – регулярное множество в алфавите Σ, то таковы же и множества:
   а) $P \cup Q$;
   б) PQ;
   в) P^*;
5) ничто другое не является регулярным множеством в алфавите Σ.

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

Определение.

Регулярные выражения в алфавите Σ и регулярные множества, которые они обозначают, определяются рекурсивно следующим образом:

1) $\varnothing$ – регулярное выражение, обозначающее регулярное множество $\varnothing$;
2) e – регулярное выражение, обозначающее регулярное множество $\{e\}$;
3) если $a \in \Sigma$, то a – регулярное выражение, обозначающее регулярное множество $\{a\}$;
4) если p и q – регулярные выражения, обозначающие регулярные множества P и Q, то
   а) $(p+q)$ – регулярное выражение, обозначающее $P \cup Q$;
   б) pq – регулярное выражение, обозначающее PQ;
   в) $(p)^*$ - регулярное выражение, обозначающее P^*;
5) ничто другое не является регулярным выражением.

Принято обозначать p^+ для сокращенного обозначение pp^*. Расстановка приоритетов:

- * (итерация)- наивысший приоритет;
- конкатенация;
- +.

Таким образом, $0 + 10^* = (0 + (1\ (0^*)))$

Пример.

01 означает {01}

0^* $\{0^*\}$

$(0+1)^*$ $\{0, 1\}^*$

$(0+1)^*$ 011 – означает множество всех цепочек, составленных из 0 и 1 и оканчивающихся цепочкой 011.

$(a+b)$ $(a+b+0+1)^*$ означает множество всех цепочек $\{0,1,a,b\}^*$, начинающихся с a или b.

$(00+11)^*$ $((01+10)(00+11)^*$ $(01+10)(00+11)^*)$ обозначает множество всех цепочек нулей и единиц, содержащих четное число 0 и чётное число 1. Таким образом для каждого регулярного множества можно найти регулярное выражение, его обозначающее, и наоборот.

Введем *леммы*, обозначающие основные алгебраические свойства регулярных выражений.

Пусть α, β и γ регулярные выражения, тогда:

1) $\alpha + \beta = \beta + \alpha$
2) $\varnothing^* = e$
3) $\alpha + (\beta + \gamma) = (\alpha+\beta)+\gamma$
4) $\alpha(\beta\gamma)=(\alpha\beta)\gamma$
5) $\alpha(\beta+\gamma)=\alpha\beta+\alpha\gamma$

6) $(\alpha+\beta)\gamma=\alpha\gamma+\beta\gamma$
7) $\alpha e = e\alpha = \alpha$
8) $\varnothing\alpha=\alpha\varnothing=\varnothing$
9) $\alpha^*=\alpha+\alpha^*$
10) $(\alpha^*)^*=\alpha^*$
11) $\alpha+\alpha=\alpha$
12) $\alpha+\varnothing=\alpha$.

При работе с языками часто удобно пользоваться уравнениями, коэффициентами и неизвестными которых служат множества. Такие уравнения будем называть уравнениями с регулярными коэффициентами

$$X = aX + b,$$

где *a* и *b*– регулярные выражения. Можно проверить прямой подстановкой, что решением этого уравнения будет a^*b.

$$a^*b = aa^*b + b,$$

т.е. получаем одно и то же множество. Таким же образом можно установить и систему уравнений.

Определение. Систему уравнений с регулярными коэффициентами назовём *стандартной системой* с множеством неизвестных $\Delta = \{X_1, X_2, ..., X_n\}$, если она имеет вид:

$$X_1 = \alpha_{10} + \alpha_{11}X_1 + \alpha_{12}X_2 + ... + \alpha_{1n}X_n$$

.....

$$X_n = \alpha_{n0} + \alpha_{n1}X_1 + \alpha_{n2}X_2 + ... + \alpha_{nn}X_n,$$

где α_{ij} – регулярные выражения в алфавите, не пересекающемся с Δ.

Коэффициентами уравнения являются выражения α_{ij}.

Если $\alpha_{ij}=\varnothing$, то в уравнении нет числа, содержащего X_j. Аналогично, если $\alpha_{ij}=e$, то в уравнении для X_i член, содержащий X_j - это просто X_j. Иными словами, $\varnothing$ играет роль коэффициента 0, а *e* – роль коэффициента 1 в обычных линейных уравнениях.

Алгоритм решения стандартной системы уравнений с регулярными выражениями.

Вход. Стандартная система Q уравнений с регулярными коэффициентами в алфавите Σ и множеством неизвестных $\Delta = \{X_1, X_2, ..., X_n\}$.

Выход. Решение системы Q .

Метод: Аналог метода решения системы линейных уравнений методом исключения Гаусса.

Шаг 1. Положить $i = 1$.

Шаг 2. Если $i = n$, перейти к шагу 4. В противном случае с помощью тождеств леммы записать уравнения для X_i в виде $X_i = \alpha X_i + \beta$, где α - регулярное выражение в алфавите Σ, а β - регулярное выражение вида: $\beta_0 + \beta_i X_{i+1} + ... + \beta_n X_n$, причём все β_i – регулярные выражения в алфавите Σ. Затем в правых частях для уравнений $X_{i+1} .... X_n$ заменим X_i регулярным выражением $\alpha^* \beta$.

Шаг 3. Увеличить i на 1 и вернуться к шагу 2.

Шаг 4. Записать уравнение для X_n в виде $X_n = \alpha X_n + \beta$, где α и β - регулярные выражения в алфавите Σ. Перейти к шагу 5 (при этом $i=n$).

Шаг 5. Уравнение для X_i имеет вид $X_i = \alpha X_i + \beta$, где α и β - регулярные выражения в алфавите Σ. Записать на выходе $X_i = \alpha^* \beta$ в уравнениях для $X_{i-1}, ..., X_1$ подставляя $\alpha^* \beta$ вместо X_i.

Шаг 6. Если i=1, остановиться, в противном случае уменьшить i на 1 и вернуться к шагу 5.

Однако следует отметить, что не все уравнения с регулярными коэффициентами обладают единственным решением. Например, если

$$X = \alpha X + \beta$$

- уравнение с регулярными коэффициентами и α означает множество, содержащее пустую цепочку, то $X = \alpha^* (\beta + \gamma)$ будет решением этого уравнения для любого γ. Таким образом, уравнение имеет бесконечно много решений. В такого рода ситуациях мы будем брать наименьшее решение, которое назовем *наименьшей неподвижной точкой*. В нашем случае наименьшая неподвижная точка $\alpha^* \beta$.

Лемма.

Каждая стандартная система уравнений Q с неизвестными Δ обладает единственной наименьшей неподвижной точкой.

Определение.

Пусть Q – стандартная система уравнений с множеством неизвестных $\Delta = \{X_1, X_2, ..., X_n\}$ в алфавите Σ. Отображение f множества Δ во

множество языков в алфавите Σ называется решением системы Q, если после подстановки в каждое уравнение $f(x)$ вместо X для каждого $X \in \Delta$ уравнения становятся равенствами множеств.

Отображение $f: \Delta \to \mathbf{P}(\Sigma^*)$ называется наименьшей неподвижной точкой системы Q, если f решение, и для любого другого решения g $f(x) \le g(x)$ для всех $X \in \Delta$..

Лемма. Пусть Q – стандартная система уравнений с неизвестными $\Delta = \{X_1, X_2, ..., X_n\}$, и уравнение для X_i имеет вид

$$X_i = \alpha_{i0} + \alpha_{i1} X_1 + ... \alpha_{in} X_n .$$

Тогда наименьшей неподвижной точкой системы Q будет такое отображение

$$f(X_i) = \{w_1, ... w_m \mid w_m \in \alpha_{j_m 0} \text{ и } w_k \in \alpha_{j_k j_{k+1}} \text{ для } j_1 ..., j_m\}$$

для некоторой последовательности чисел $j_1, j_2, ..., j_m$, где $m \ge 1$, $1 \le k \le m$, $j_1 = i$.

Примем в качестве аксиомы утверждение, что язык определяется праволинейной грамматикой тогда и только тогда, когда он является регулярным множеством. Таким образом, констатируем:

1) класс регулярных множеств – наименьший класс языков, содержащих множества $\varnothing$, $\{e\}$ и $\{a\}$ для всех символов a и замкнутый относительно операций объединения, конкатенации и итерации;
2) регулярные множества – множества, определённые регулярными выражениями;
3) регулярные множества – языки, порождаемые праволинейными грамматиками.

3.6. Регулярные множества и конечные автоматы

Ещё одним удобным способом определения регулярных множеств являются конечные автоматы.

Качественное описание конечных автоматов мы сделали в разделе 3.4. Теперь дадим их математическую формулировку.

Определение. Недетерминированный конечный автомат – это пятёрка $M = (Q, \Sigma, \delta, q_0, F)$,

где Q – конечное множество состояний;

Σ - конечное множество допустимых входных символов;

δ - отображение множества $\Sigma \times Q$ во множество $\mathbf{P}(Q)$, определяющее поведение управляющего устройства, его обычно называют функцией переходов;

$q_0 \in Q$ - начальное состояние управляющего устройства;

$F \subseteq Q$ - множество заключительных состояний.

Работа конечного автомата представляет собой некоторую последовательность шагов (тактов). Такт определяется текущим состоянием управляющего устройства и входным символом, обозреваемым в настоящий момент входной головкой. Сам шаг состоит из изменения состояния и сдвига входной головки на одну ячейку вправо. Для того чтобы определить будущее поведение конечного автомата, нужно знать:

1) текущее состояние управляющего устройства;
2) цепочку символов на входной ленте, состоящую из символа под головкой и всех символов, расположенных вправо от него.

Эти два элемента информации дают мгновенное описание конечного автомата, которое называется конфигурацией.

Определение.

Если $M = (Q, \Sigma, \delta, q_0, F)$ – конечный автомат, то пара $(q \times w) \in Q \times \Sigma^*$ называется конфигурацией автомата M.

Конфигурация $(q_0 \times w)$ называется *начальной*, а пара (q, e), где $q \in F$, называется *заключительной*.

Такт автомата M представляется бинарным отношением $\vdash_M$, определённым на конфигурациях. Это говорит о том, что, если M находится в состоянии q и входная головка обозревает символ a, то автомат M может делать такт, за который он переходит в состояние q' и сдвигает головку на одну ячейку вправо. Так как автомат M, вообще говоря, недетерминирован, могут быть и другие состояния, отличные от q', в он может перейти за один шаг.

Запись $C \vdash_M^0 C'$ означает, что $C = C'$, а $C^0 \vdash_M^\kappa C_k$ (для $\kappa \geq 1$) – что существует конфигурация $C_1, \ldots C_{\kappa-1}$, такая что $C_i \vdash_M C_{i+1}$ для всех $0 \leq i < k$. $C \vdash_M^+ C'$ означает, что $C \vdash_M^\kappa C'$ для некоторого $\kappa \geq 0$. Таким образом, отношения $\vdash_M^+$ и $\vdash_M^*$ являются транзитивным и рефлексивно-транзитивным замыканием отношения $\vdash_M$.

Говорят, что M допускает цепочку w, если $(q_0, w) \vdash^* (q, e)$ для некоторого $q \in F$.

Языком, определяемым автоматом M ($L(M)$), называется множество входных цепочек, допускаемых автоматом M, т.е.

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

Пример 1:

Пусть $M = (\{p, q, r\}, \{0, 1\}, \delta, p, \{r\})$ конечный автомат, где δ задаётся в виде таблицы 3.1.

Таблица 3.1 – Таблица состояний конечного автомата

δ	Вход	
	0	1
Состояние p	$\{q\}$	$\{p\}$
q	$\{r\}$	$\{p\}$
r	$\{r\}$	$\{r\}$

M допускает все цепочки нулей и единиц, содержащих два стоящих рядом 0. Начальное состояние p можно интерпретировать как «два стоящих рядом нуля ещё не появились и предыдущий символ не был нулём». Состояние q означает, что «два стоящих рядом нуля ещё не появились, но предыдущий символ был нулём». Состояние r означает, что «два стоящих рядом нуля уже появились», т.е. попав в состояние r автомат остаётся в этом состоянии.

Для входа 01001 единственной возможной последовательностью конфигураций, начинающейся конфигурацией $(p, 01001)$, будет

$$(p, 01001) \vdash (q, 1001)$$
$$\vdash (p,001)$$
$$\vdash (q,01)$$
$$\vdash (r,1)$$
$$\vdash (r, e).$$

Таким образом $01001\in L(M)$.

Пример 2.

Построим недетерминированный конечный автомат, допускающий цепочки алфавита $\{1,2,3\}$, у которого последний символ цепочки уже появлялся раньше. Иными словами 121 допускается, а 31312 – нет. Введем состояние q_0, смысл которого в том, что автомат в этом состоянии не пытается ничего распознать (начальное состояние). Введем состояния q_1, q_2 и q_3, смысл которых в том, что они «делают предположение» о том, что последний символ цепочки совпадает с индексом состояния. Кроме того, пусть будет одно заключительное состояние q_f. Находясь в состоянии q_0, автомат может остаться в нем или перейти в состояние q_a, если a – очередной символ (табл. 3.2). Находясь в со-

стоянии q_a автомат может перейти в состояние q_f , если видит символ a.

Таблица 3.2 – Таблица состояний конечного автомата

δ	Вход		
	1	2	3
Состояние q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_3\}$
q_1	$\{q_1, q_f\}$	$\{q_1\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2, q_f\}$	$\{q_2\}$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_f\}$
q_f	$\varnothing$	$\varnothing$	$\varnothing$

Формально автомат M определяется как пятерка

$$M = \left(\{q_0, q_1, q_2, q_f\}, \{1,2,3\}, \delta, q_0, \{q_f\}\right).$$

Часто удобно графическое представление конечного автомата.

3.7. Графическое представление конечных автоматов

Определение. Пусть $M = (Q, \Sigma, \delta, q_0, F)$ – недетерминированный конечный автомат. Диаграммой δ переходов (графом переходов) автомата M называется неупорядоченный помеченный граф, вершины которого помечены именами состояний и в котором есть дуга (p, q), если существует такой символ $a \in \Sigma$, что $q \in \delta(p, a)$. Кроме того, дуга (p,q) помечается списком, состоящим из таких a, что $q \in \delta(p, a)$. Изобразим автоматы из предыдущих примеров в виде графов (рис. 3.2 и 3.3)

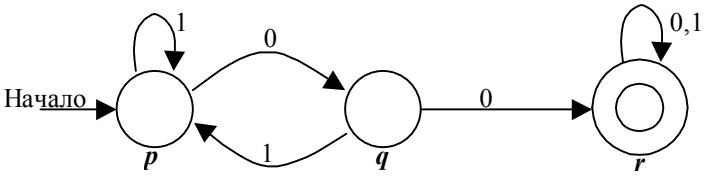


Рис. 3.2. Пример детерминированного графа

Для дальнейшего анализа нам потребуется определение детерминированного автомата.

Определение.

Пусть $M = (Q, \Sigma, \delta, q_0, F)$ – недетерминированный конечный автомат. Назовём автомат M детерминированным, если множество $\delta(q, a)$ содержит не более одного состояния для любых $q \in Q$ и $a \in \Sigma$. Если $\delta(q,a)$ всегда содержит точно одно состояние, то автомат M назовём полностью определённым.

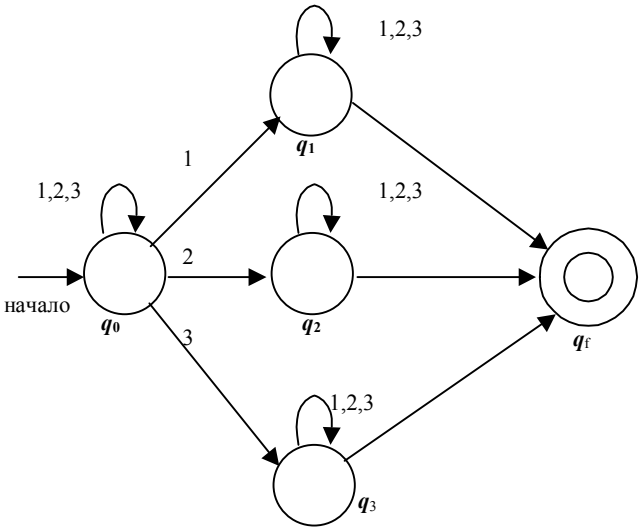


Рис. 3.3. Пример недетерминированного графа

Таким образом, наш пример – полностью определённый детерминированный конечный автомат, и в дальнейшем под конечным автоматом мы будем подразумевать полностью определённый конечный автомат.

Одним из важнейших результатов теории конечных автоматов является тот факт, что класс языков, определяемых недетерминированными конечными автоматами, совпадает с классом языков, определяемых детерминированными конечными автоматами.

Теорема.

Если $L=L(M)$ для некоторого недетерминированного конечного автомата, то $L=L(M')$ для некоторого конечного автомата M'.

Пусть $M = (Q, \Sigma, \delta, q_0, F)$. Построим автомат $M' = (Q', \Sigma, \delta', q_0', F')$ следующим образом:

1) $Q' = \mathsf{P}(Q)$, т.е. состояниями автомата M' является множество состояний автомата M;
2) $q_0' = \{q_0\}$;
3) F' состоит из всех таких подмножеств S множества Q, что $S \cap F \neq \varnothing$;
4) $\delta(S, a) = S'$ для всех $S \subseteq Q$, где $S' = \{p \mid \delta(q, a)$ содержит p для некоторого $q \in S\}$.

Пример.

Построим конечный автомат $M' = (Q, \{1, 2, 3\}, \delta', \{q_0\}, F)$, допускающий язык $L(M)$.

Так как M имеет 5 состояний, то в общем случае M' должен иметь 32 состояния. Однако, не все они достижимы из начального состояния. Состояние p называется достижимым, если существует такая цепочка w, что $(q_0, w) \vdash^* (p, e)$, где q_0 - начальное состояние. Мы будем строить только достижимые состояния (табл. 3.3).

Таблица 3.3 - Достижимые состояния автомата

Состояние	***Вход***		
	1	***2***	***3***
$A = \{q_0\}$	B	C	D
$B = \{q_0, q_1\}$	E	F	G
$C = \{q_0, q_2\}$	F	H	I
$D = \{q_0, q_3\}$	G	I	J
$E = \{q_0, q_1, q_f\}$	E	F	G
$F = \{q_0, q_1, q_2\}$	K	K	L
$G = \{q_0, q_1, q_3\}$	M	L	M
$H = \{q_0, q_2, q_f\}$	F	H	I
$I = \{q_0, q_2, q_3\}$	L	N	N
$J = \{q_0, q_3, q_f\}$	G	I	J
$K = \{q_0, q_1, q_2, q_f\}$	K	K	L
$L = \{q_0, q_1, q_2, q_3\}$	P	P	P
$M = \{q_0, q_1, q_3, q_f\}$	M	L	M
$N = \{q_0, q_2, q_3, q_f\}$	L	N	N
$P = \{q_0, q_1, q_2, q_3, q_f\}$	P	P	P

Начнём с замечания, что состояние $\{q_0\}$ достижимо. $\delta'(\{q_0\}, a) = \{q_0, q_a\}$ для $a = 1, 2, 3$. Рассмотрим состояние $\{q_0, q_1\}$. Имеем $\delta'(\{q_0, q_1\}, 1) = \{q_0, q_1, q_f\}$. Продолжая, по данной схеме, получаем, что множество состояний автомата M (M') достижимо тогда и только тогда, когда

1) оно содержит q_0 и

2) если оно содержит q_f, то содержит также и q_1, q_2 или q_3.

Начальным состоянием автомата M является A, а множество заключительных состояний - $\{E, H, J, K, M, N, P\}$.

3.8. Конечные автоматы и регулярные множества

Из теории регулярных множеств и конечных автоматов можно доказать, что язык является регулярным множеством тогда и только тогда, когда он является конечным автоматом. Это следует из следующих лемм, принимаемых нами без доказательств.

Лемма. Если $L=L(M)$ для некоторого конечного автомата, то $L=L(G)$ для некоторой праволинейной грамматики.

Лемма. Пусть Σ - конечный алфавит, тогда множества (1) $\varnothing$, (2) $\{e\}$ и $\{a\}$ для всех $a \in \Sigma$ являются конечно-автоматными языками.

Лемма. Пусть $L_1 = L(M_1)$ и $L_2 = L(M_2)$ для конечных автоматов M_1 и M_2. Множества $L_1 \cup L_2$, $L_1 L_2$ и L_1^* являются конечно-автоматными языками.

Заключительная теорема.

Язык допускается конечным автоматом тогда и только тогда, когда он является регулярным множеством. Таким образом утверждения

1) L – регулярное множество;
2) L – праволинейный язык;
3) L – конечно-автоматный язык

эквивалентны.

3.9. Минимизация конечных автоматов

Цель. Показать, что для каждого регулярного множества однозначно находится конечный автомат с минимальным числом состояний.

По данному конечному автомату M можно найти наименьший эквивалентный ему конечный автомат, исключив все недостижимые состояния и затем склеив лишнее состояние. Лишнее состояние определяется с помощью разбиения множества всех состояний на классы эквивалентности так, что каждый класс содержит неразличимые состояния и выбирается как можно шире. Потом из каждого класса берётся один представитель в качестве состояния сокращенного (или приведённого) автомата.

Таким образом можно сократить объём автомата M, если M содержит недостижимые состояния или два или более неразличимые состояния. Полученный автомат будет наименьший из конечных

автоматов, распознающий регулярное множество, определяемое первоначальным автоматом M.

Определение.

Пусть $M = (Q, \Sigma, \delta, q_0, F)$ конечный автомат, а q_1 и q_2 два его различные состояния. Будем говорить, что цепочка $x \in \Sigma^*$ различает состояния q_1 и q_2, если $(q_1, x) \vdash^* (q_3, e)$, $(q_2, x) \vdash^* (q_4, e)$ и одно из состояний q_3 и q_4 принадлежит F. Будем говорить, что q_1 и q_2 k- неразличимы, и писать $q_1 \equiv^k q_2$, если не существует такой цепочки x, различающей q_1 и q_2, у которой $|x| \le k$. Будем говорить, что состояния q_1 и q_2, неразличимы и писать $q_1 \equiv q_2$, если они k – неразличимы для любого $k \ge 0$.

Состояние $q \in Q$ называется недостижимым, если не существует такой входной цепочки x, что $(q_0, x) \vdash^* (q, e)$.

Автомат M называется приведенным, если в Q нет недостижимых состояний и нет двух неразличимых состояний.

Пример. Рассмотрим конечный автомат M с диаграммой (рис. 3.4).

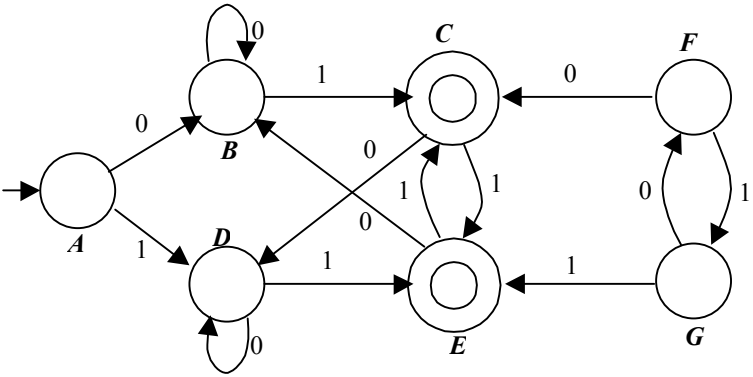


Рис. 3.4. Диаграмма автомата M

Чтобы сократить M, заметим сначала, что состояния F и G недостижимы из начального состояния A, так что их можно устранить. Пока качественно, а позже строго мы установим, что классами эквивалентности отношений являются $\{A\}$, $\{B, D\}$ и $\{C, E\}$. Тогда, взяв представителями этих множеств p, q и r, можно получить конечный автомат вида (рис. 3.5).

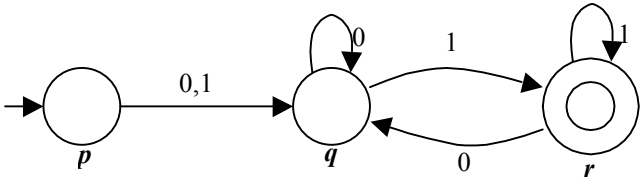


Рис. 3.5. Приведенный конечный автомат

Перед тем, как построить алгоритм канонического конечного автомата, введём лемму:

Лемма.

Пусть $M = (Q, \Sigma, \delta, q_0, F)$ конечный автомат с n состояниями. Состояния q_1 и q_2 неразличимы тогда и только тогда, когда они $(n\text{-}2)$ – неразличимы, т.е. если два состояния можно различить, то их можно различить с помощью входной цепочки, длина которой меньше числа состояний автомата.

Алгоритм построения канонического конечного автомата.

Вход.

Конечный автомат $M = (Q, \Sigma, \delta, q_0, F)$.

Выход.

Эквивалентный конечный автомат M'.

Метод.

Шаг 1. Применив к диаграмме автомата M алгоритм нахождения множества вершин, достижимых из данной вершины ориентированного графа, найти состояния достижимые из q_0. Установить все недостижимые состояния.

Шаг 2. Строить отношения эквивалентности $\equiv^0$, $\equiv^1$ по схеме:

1) $q_1 \equiv^0 q_2$ тогда и только тогда, когда они оба принадлежат, либо оба не принадлежат F;
2) $q_1 \equiv^k q_2$ тогда и только тогда, когда $q_1 \equiv^{k-1} q_2$ и $\delta(q_1, a) \equiv^k \delta(q_2, a)$ для всех $a \in \Sigma$.

Шаг 3. Построить конечный автомат $M'=(Q', \Sigma, \delta', q_0', F')$,

где Q' – множество классов эквивалентности отношений $\equiv$ (обозначим через $[p]$ класс эквивалентности отношений, содержащий состояние p);

$(\delta', [p], a) = [q]$, если $\delta(p, a) = q$;

q_0' – это $[q_0]$;

F' - $\{[q] \mid q \in F\}$.

Теорема.

Автомат *M*', построенный по данному алгоритму имеет наименьшее число состояний среди всех конечных автоматов, допускающих язык *L*(*M*).

Пример.

Найдём приведённый конечный автомат автомату *M* (рис. 3.6).

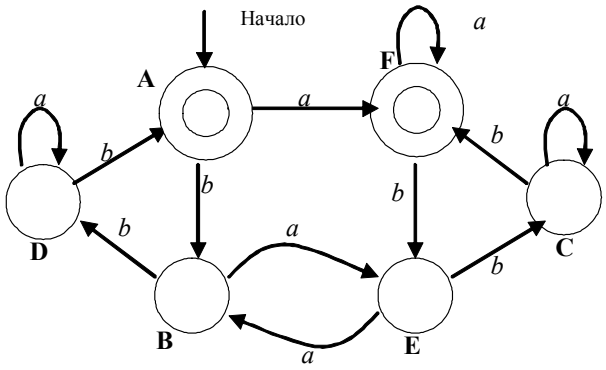


Рис. 3.6. Диаграмма автомата *M*

Отношения $\equiv^k$ для $\kappa \geq 0$ имеют следующие классы эквивалентности:

класс отношения $\equiv^0$ {A, F}, {B, C, D, E};

класс отношения $\equiv^1$ {A, F}, {B, E}, {C, D};

класс отношения $\equiv^2$ {A,F}, {B,E}, {C,D}.

Так как $\equiv^2 = \equiv^1$, то $\equiv = \equiv^1$.Приведённый автомат *M′* будет ({[A],[B], [C], {*a, b*}, δ', A, {[A]}}), где δ' определяется следующей таблицей.

Таблица 3.4 - Приведенный конечный автомат

Состояние	*a*	*b*
[A]	[A]	[B]
[B]	[B]	[C]
[C]	[C]	[A]

[A] – выбрано для представления класса {A, F};

[B] – выбрано для представления класса {B, E};

[C] – выбрано для представления класса {C, D}.

3.10. Контекстно-свободные языки

Из четырёх классов грамматики иерархии Хомского класс контекстно-свободных языков наиболее важен с точки зрения приложения к языкам программирования и компиляции. С их помощью можно определить большую часть синтаксических структур языков программирования. Кроме того, они служат основой различных схем перевода, т.к. в ходе процесса компиляции синтаксическую структуру, передаваемую входной программе КС-грамматикой, можно использовать при построении перевода этой программы.

Синтаксическую структуру входной цепочки можно определить по последовательности правил, применяемых при выводе этой цепочки. Таким образом, на часть компилятора, называемую синтаксическим анализатором, можно смотреть как на устройство, которое пытается выяснить, существует ли в некоторой фиксированной КС-грамматике вывод входной цепочки.

Естественно, что эта задача нетривиальная – по данной КС-грамматике G и входной цепочке *w* выяснить, принадлежит ли *w* языку $L(G)$, и если «да», то найти вывод цепочки *w* в грамматике *G*.

3.10.1. Деревья выводов

В грамматике может быть несколько выводов, эквивалентных в том смысле, что во всех них применяются одни и те же правила в одних и тех же местах, но в различном порядке. КС-грамматика позволяет ввести удобное графическое представление класса эквивалентных выводов, называемое деревом выводов.

Определение.

Дерево вывода в КС-грамматике $G = (N, \Sigma, P, S)$,

где N – конечное множество нетерминальных символов;

Σ - непересекающееся с N множество терминальных символов;

P – конечное подмножество множества $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ (элемент (α, β) множества P называется правилом или продукцией $\alpha \to \beta$);

S – выделенный символ из N, называемый начальным или исходным символом,

- это помеченное упорядоченное дерево, каждая вершина которого помечена символом из множества $N \cup \Sigma \cup \{e\}$.

Если внутренняя вершина помечена символом A, а её прямые потомки - символами $X_1, X_2, ..., X_n$, то A$\rightarrow$ $X_1, X_2, ..., X_n$ – правило грамматики.

Определение.

Помеченное упорядоченное дерево D называется деревом вывода (или деревом разбора) в КС-грамматике G(A) = (N, Σ, P, A), если выполняются следующие условия:

1) корень дерева помечен A;
2) если $D_1, D_2, ..., D_k$ – поддеревья, над которыми доминируют прямые потомки корня дерева, и корень D_i помечен X_i, то A$\rightarrow$ $X_1, X_2, ..., X_n$ – правило из множества P. D_i должно быть деревом вывода в грамматике $G(X_i)$ = (N, Σ, P, X_i), если X_i – нетерминал, и D_i состоит из единственной вершины, помеченной X_i, если X_i - терминал.

Пример. Имеем грамматику G=G(S) с правилами S$\rightarrow$*a*S*b*S | *b*S*a*S | *e* (рис.3.7).

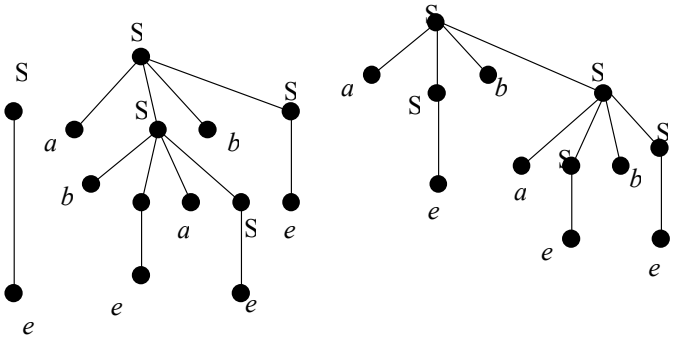


Рис. 3.7. Деревья выводов

$$\left.\begin{array}{l} S \rightarrow aSba \\ S \rightarrow bSab \end{array}\right\} \quad S\rightarrow aSbS \mid bSaS \mid e$$
$$S \rightarrow e$$

Заметим, что существует единственное упорядочение вершин упорядоченного дерева, у которого прямые потомки вершины упорядочиваются «слева направо».

Допустим, что n – вершина и $n_1, n_2, ..., n_k$ – её прямые потомки. Тогда если $i<j$, то вершина n_i и все её потомки считаются расположенными левее вершины n_i и всех её потомков.

Определение.

Кроной дерева вывода назовём цепочку, которая получится, если выписать слева направо метки листьев.

Кроны наших деревьев: S, *e*, *abab*, *abab*.

Определение.

Сечением дерева D назовём такое множество C вершин дерева D, что

1) никакие две вершины из C не лежат на одном пути в D;
2) ни одну вершину дерева D нельзя добавить к C, не нарушив свойства 1).

Пример.

- Множество вершин дерева, состоящего из одного корня, является сечением.
- Листья также образуют сечение.

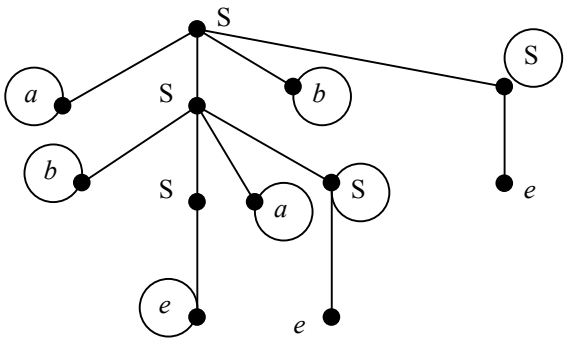


Рис. 3.8. Пример сечения дерева

Определение.

Кроной сечения дерева D является цепочка, получаемая конкатенацией (в порядке слева направо) меток вершин, образующих некоторое сечение.

Крона сечения примера, приведенного на рис.3.7, - *aba*S*b*S.

Лемма.

Пусть S=$\alpha_1,\alpha_2,...,\alpha_n$ – вывод цепочки α_n из S в КС – грамматике G = (N, Σ, P, S). Тогда в G можно построить дерево вывода D, для которого α_n – крона, а $\alpha_1,\alpha_2,...,\alpha_{n-1}$ – некоторые из крон сечения.

Лемма.

Пусть D – дерево вывода в КС-грамматике G = (N, Σ, P, S) с кроной α. Тогда S$\Rightarrow^*$α (транзитивное и рефлексивное замыкание $\Rightarrow^*$, т.е. α выводима из S).

Доказательство.

Пусть $C_0, C_1, ..., C_n$ – такая последовательность сечений дерева D, что

1) C_0 – содержит только один корень дерева D;
2) C_{i+1} для $0 \le i < n$ получается из C_i заменой одной нетерминальной вершины её прямыми потомками;
3) C_n – крона дерева D.

Ясно, что хотя бы одна такая последовательность существует.

Если α_i – крона сечения C_i, то существующий вывод $\alpha_1,\alpha_2,...,\alpha_n$ называется левым выводом цепочки α_n из α_0 в грамматике G. Правый вывод определяется аналогично.

Если S=$\alpha_1,\alpha_2,...,\alpha_{n-1}$ = w – левый вывод терминальной цепочки w, то каждая цепочка α_i $(0 \le i < n)$ имеет вид $x_i A_i \beta_i$, где $x_i \in \Sigma^*$, $A_i \in$N и $\beta_i \in (N \cup \Sigma)^*$.

Каждая следующая цепочка α_{i+1} левого вывода получается из предыдущей цепочки α_i заменой самого левого нетерминала A_i правой частью некоторого правила.

Пример. Рассмотрим КС-грамматику G_0 с правилами

E→E+T | T
T→T*F | F
F→(E) | *a*.

Нарисуем дерево вывода (рис 3.7).

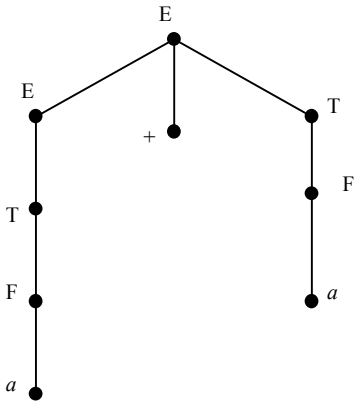


Рис. 3.9. Пример дерева вывода

Это дерево служит представлением десяти эквивалентных выводов цепочки *a*+*a*.

E⇒ E+T ⇒T+T ⇒F+T⇒ *a*+T⇒ *a*+F⇒*a*+*a* – левый вывод.

E⇒ E+T ⇒E+F ⇒E+*a* ⇒ T+*a*⇒ F+*a*⇒*a*+*a* – правый вывод.

Определение. Цепочку α будем называть *левовыводимой*, если существует левый вывод S=$\alpha_1,\alpha_2,...,\alpha_n$ = α, и писать S$\Rightarrow_l^* \alpha_n$. Аналогично α будем называть *правовыводимой*, если существует правый вывод S=$\alpha_1,\alpha_2,...,\alpha_n$ = α, и писать S$\Rightarrow_r^* \alpha_n$. Один шаг левого вывода будем обозначать $\Rightarrow_l$, а правого $\Rightarrow_r$

3.10.2. Преобразование КС–грамматик

КС-грамматику часто требуется модифицировать так, чтобы порождаемые ею языки приобрели нужную структуру. Рассмотрим, например, язык $L(G_0)$. Этот язык порождается грамматикой *G* с правилами

E→E+E | E*E | (E) | *a*.

Но эта грамматика имеет два недостатка. Прежде всего, она неоднозначна из-за наличия правила E→E+E | E*E. Эту неоднозначность можно устранить, взяв вместо *G* грамматику G_1 с правилами

E→E+T | E*T | T

T→(E) | *a*.

Другой недостаток грамматики *G*, которым обладает и грамматика G_1, заключается в том, что операции + и * имеют один и тот же при-

оритет, т.е. структура выражения $a+a*a$ и $a*a+a$, которую мы придаём грамматике G_1, подразумевает тот же порядок выполнения операций, что и в выражениях $(a+a)*a$ и $(a*a)+a$ соответственно.

Чтобы получить обычный приоритет операций + и *, при которых * предшествует + и выражение $a+(a*a)$ понимается как $a+(a*a)$, надо перейти к грамматике G_0.

Общего алгоритмического метода, который придавал бы данному языку произвольную структуру, не существует. Но с помощью ряда преобразований можно видоизменить грамматику, не испортив порождаемый ею язык.

Начнём с очевидных, но важных преобразований. Например, в грамматике $G=(\{S,A\}, \{a, b\}, P, S)$, где $P=\{S\to a, A\to b\}$, нетерминал A и терминал b не могут появляться ни в какой выводимой цепочке. Таким образом, эти символы не имеют отношения к языку $L(G)$ и их можно устранить из определения грамматики G, не затронув языка $L(G)$.

Определение.

Назовём символ $X\in N\cup\Sigma$ *бесполезным* в КС – грамматике $G = (N, \Sigma, P, S)$, если в ней нет вывода вида $S\Rightarrow^* wXy \Rightarrow^* wxy$, где w, x, y принадлежат Σ^*.

Чтобы установить, бесполезен ли нетерминал A, построим сначала алгоритм, выясняющий, может ли этот нетерминал порождать какие-либо нетерминальные цепочки, т.е. алгоритм, решающий проблему пустоты множества $\{w \mid A \Rightarrow^* w, w\in \Sigma^*\}$.

Алгоритм. Непуст ли язык $L(G)$?

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

Выход. «ДА» если $L(G)\neq\varnothing$, «НЕТ» в противном случае.

Метод. Строим множества $N_0, N_1, \ldots$ рекурсивно.

1) Положить $N_0 = \varnothing$, i=1.
2) Положить $N_i = \left\{A \middle| A \to \alpha \in P \text{ и } \alpha \in \left(N_{i-1} \cup \Sigma\right)^*\right\} \cup N_{i-1}$.
3) Если $N_i \neq N_{i-1}$, то положить $i=i+1$ и перейти к шагу 2), в противном случае - $N_e = N_i$.
4) если $S\in N_e$, то выдать вывод «ДА», в противном случае – «НЕТ».

Так как символ $N_e \subseteq N$, то алгоритм должен остановиться максимум после n+1 повторения шага 2).

Теорема.

Алгоритм, приведённый выше, говорит «ДА» тогда и только тогда, когда $S \Rightarrow^* w$ для некоторой цепочки $w \in \Sigma^*$.

Определение.

Символ $X \in N \cup \Sigma$ назовём недостижимым в КС – грамматике $G = (N, \Sigma, P, S)$, если x не появляется ни в одной выводимой цепочке.

Недостижимые символы можно устранить из КС – грамматики с помощью следующего алгоритма.

Алгоритм устранения недостижимых символов.

Вход. КС – грамматика $G = (N, \Sigma, P, S)$.

Выход. КС - грамматика $G' = (N', \Sigma', P', S)$, у которой

i) $L(G')=L(G)$,

ii) для всех $X \in N' \cup \Sigma'$ существуют такие цепочки α и β из $(N' \cup \Sigma')^*$, что $S \Rightarrow^*_{G'} \alpha X \beta$.

Метод.

1) Положить $V_0 = \{S\}$ и $i=1$.
2) Положить $V_i = \left\{ X \middle| \text{ в } P \text{ есть } A \to \alpha X \beta \text{ и } A \in V_{i-1} \right\} \cup V_{i-1}$.
3) Если $V_i \neq V_{i-1}$, положить $i=i+1$ и перейти к шагу (2), в противном случае, пусть
   - $N' = V_i \cap N$,
   - $\Sigma' = V_i \cap \Sigma$,
   - P' – состоит из правил множества P, содержащих только символы из V_i,
   - $G' = (N', \Sigma', P', S)$.

Заметим, что шаг 2) алгоритма можно повторить только конечное число раз, т.к. $V_i \subseteq N \cup \Sigma$.

На базе двух рассмотренных алгоритмов построим обобщенный алгоритм устранения бесполезных символов.

Алгоритм устранения бесполезных символов.

Вход. КС – грамматика $G = (N, \Sigma, P, S)$, у которой $L(G) \neq \varnothing$.

Выход. КС - грамматика $G' = (N', \Sigma', P', S)$, у которой $L(G')=L(G)$ и в $N' \cup \Sigma'$ нет бесполезных символов.

Метод.

1) Применив к G алгоритм «не пуст ли язык?», получить N_e, положить $G_1 = (N \cap N_e, \Sigma, P_1, S)$, где P_1 состоит из множества правил P, содержащих только символы из $N_e \cup \Sigma$.
2) Применив к G_1 алгоритм «устранение недостижимых символов», получить $G' = (N', \Sigma', P', S)$.

Таким образом, на шаге 1) нашего алгоритма из G устраняются все нетерминалы, которые не могут порождать терминальных цепочек. Затем на шаге 2) устраняются все недостижимые символы.

Каждый символ X результирующей грамматики должен появиться хотя бы в одном выводе вида $S \Rightarrow^* wXy \Rightarrow^* wxy$.

Резюме. Грамматика G', которую строит рассматриваемый алгоритм, не содержит бесполезных символов.

В практике построения трансляторов обычно правила $A \to e$ бессмысленны. Очень полезно отработать метод устранения таких правил из грамматики.

Определение.

Назовём КС – грамматику $G = (N, \Sigma, P, S)$ грамматикой без e - правил (или *неукорачивающей*), если либо P не содержит e – правил, либо есть точно одно правило $S \to e$ и S не встречается в правых частях остальных правил из P.

Алгоритм преобразования в грамматику без e – правил.

Вход. КС – грамматика $G = (N, \Sigma, P, S)$.

Выход. Эквивалентная КС - грамматика $G' = (N', \Sigma', P', S)$ без e – правил.

Метод.

1) Построить $N_e = \left\{ A \middle| A \in N \text{ и } A \Rightarrow_G^+ e \right\}$.
2) Построить P′ следующим образом:

а) если $A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 ... B_k \alpha_k$ принадлежит P, $k \geq 0$ и $B_i \in N_e$ для $1 \leq i \leq \kappa$, но ни один символ в цепочках α_j $(0 \leq j \leq k)$ не принадлежит N_e, то включить в P′ все правила вида:

$$A \to \alpha_0 X_1 \alpha_1 X_2 ... \alpha_{k-1} X_k \alpha_k ,$$

где X_i - либо B_i, либо e, но не включает правило $A \to e$ (это могло бы произойти в случае, если все α_i равны e);

б) если $S \in N_e$, включить в P′ правила S′→e | S, где S′ – новый символ, и положить N′=N∪{S′}, в противном случае положить N′=N и S′=S.

3) Положить G' = (N′, Σ′, P′, S′).

Пример.

Рассмотрим грамматику S→aSbS | bSaS | e. Применяя к ней рассмотренный алгоритм, получаем грамматику

S'→S | e

S→ aSbS | bSaS | aSb | abS | ab | bSa | baS | ba.

Другое полезное преобразование грамматик – устранение правил вида $A \to B$, которые мы будем называть цепными.

Алгоритм устранения цепных правил.

Вход. КС – грамматика G = (N, Σ, P, S) без e – правил.

Выход. Эквивалентная КС - грамматика G' = (N′, Σ′, P′, S) без e – правил и цепных правил.

Метод.

1) Для каждого A∈N построить $N_A = \{B \mid A \Rightarrow^* B\}$ следующим образом.

а) положить $N_0 = \{A\}$ и i=1;

б) положить $N_i = \left\{C \middle| B \to C \text{ принадлежит } P \text{ и } B \in N_{i-1}\right\} \cup N_{i-1}$;

в) если $N_i \neq N_{i-1}$, то положить i=i+1 и повторить шаг б), в противном случае положить $N_A = N_i$.

2) Построить P′ следующим образом: если B→α принадлежит P и не является цепным правилом, включать в P′ правило A→α для таких A, что $B \in N_A$.

3) Положить G' = (N′, Σ′, P′, S).

Пример.

Грамматика с правилами

E → E+T | T

T → T * F | F

F → (E) | a.

Применим к данной грамматике рассмотренный выше алгоритм. На шаге 1) N_E = {E, T, F}, N_T = {T, F}, N_F = {F}. После шага 2) множество P' станет такими:

E → E+T | T*F | (E)| a

T → T*F | (E) | a

F → (E) | a.

3.10.3. Грамматика без циклов

Определение.

КС – грамматика $G = (N, \Sigma, P, S)$ называется грамматикой без циклов, если в ней нет вывода $A \Rightarrow^{+} A$ для $A \in N$. Грамматика называется *приведённой*, если она без циклов, без e – правил и без бесполезных символов.

Большинство (если не все) языки программирования обладают именно этими свойствами.

3.10.4. Нормальная форма Хомского

Определение. КС–грамматика $G = (N, \Sigma, P, S)$ называется грамматикой в *нормальной форме Хомского* (или *бинарной нормальной форме*), если каждое правило из P имеет один из следующих видов:

1) $A \to BC$, где A, B, C принадлежит N;
2) $A \to a$, где $a \in \Sigma$;
3) $S \to e$, если $e \in L(G)$, причём S не встречается в правых частях правил.

На практике каждый КС – язык порождается грамматикой в нормальной форме Хомского. Это очень полезно, когда требуется простая форма представления КС – языка.

Алгоритм преобразования к нормальной форме Хомского.

Вход. КС–грамматика $G = (N, \Sigma, P, S)$.

Выход. Эквивалентная КС-грамматика G' в нормальной форме Хомского, т.е. $L(G')=L(G)$.

Метод. Грамматика G' строится следующим образом.

1) Включить в P' каждое правило из P вида $A \to a$.
2) Включить в P' каждое правило из P вида $A \to BC$.
3) включить в P' каждое правило $S \to e$, если оно было в P.
4) для каждого правила из P вида $A \to X_1 \ldots X_k$, где $k > 2$, включить в P' правила

$$A \to X_1' \langle X_2 \ldots X_k \rangle$$
$$\langle X_2 \ldots X_k \rangle \to X_2' \langle X_3 \ldots X_k \rangle$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$\langle X_{k-2} X_{k-1} X_k \rangle \to X_{k-2}' \langle X_{k-1} X_k \rangle$$
$$\langle X_{k-1} X_k \rangle \to X_{k-1}' X_k',$$

где $X_i' = X_i$, если $X_i \in N$; X_i' – новый нетерминал, если $X_i \in \Sigma$; $\langle X_i \dots X_k \rangle$ - новый терминал.

5) Для каждого правила из P вида $A \to X_1 X_2$, где хотя бы один из символов X_1 и X_2 принадлежит Σ, включить в P′ правило $A \to X_1' X_2'$.

6) Для каждого нетерминала вида *a*', введённого на шагах 4) и 5), включить в P′ правило $a' \to a$. Наконец, пусть N′ – это N вместе со всеми нетерминалами, введёнными при построении P'. Тогда искомая грамматика G' = (N′, Σ′, P′, S).

Пример.

Пусть G – приведённая грамматика, определённая правилами

$$S \to aAB \mid BA$$
$$A \to BBB \mid a$$
$$B \to AS \mid b.$$

Строим P′ рассмотренным выше алгоритмом, сохраняя правила $S \to BA$, $A \to a$, $B \to AS$ и $B \to b$. Заменяем правило $S \to aAB$ на $S \to a'\langle AB\rangle$ и $\langle AB\rangle \to AB$, а $A \to BBB$ – правилами $A \to B\langle BB\rangle$ и $\langle BB\rangle \to BB$. Наконец, добавляем $a' \to a$. В результате получим грамматику G'= (N′, {*a*, *b*}, P′, S) и P' состоит из правил:

$$S \to a'\langle AB\rangle \mid BA$$
$$A \to B\langle BB\rangle \mid a$$
$$B \to AS \mid b$$
$$\langle AB\rangle \to AB$$
$$\langle BB\rangle \to BB$$
$$a' \to a.$$

3.10.5. Нормальная формула Грейбах

Очень важно для языков программирования использовать грамматику, в которой все правые части правил начинаются с терминалов. Построение таких грамматик связано с устранением любой рекурсии.

Определение.

Нетерминал *A* в КС–грамматике *G* = (N, Σ, P, S) называется рекурсивным, если $A \Rightarrow^{+} \alpha A \beta$ для некоторых α и β. Если α = *e*, то A называется *леворекурсивным*. Аналогично, если β = *e*, то A называется *праворекурсивным*. Грамматика, имеющая хотя бы один леворекурсивный терминал, называется леворекурсивной. Аналогично определяется и праворекурсивная грамматика. Грамматика, в которой все не

терминалы, кроме может быть начального символа, рекурсивные называется рекурсивной грамматикой.

Практически все языки программирования определяются нелеворекурсивной грамматикой. Поэтому все элементы левой рекурсии должны быть устранены из грамматики.

Лемма.

Пусть G = (N, Σ, P, S) – КС–грамматика, в которой

$$A = A\alpha_1 \left| A\alpha_2 \right| \ldots A\alpha_m \left| \beta_1 \right| \beta_2 \left| \ldots \beta_n \right.$$

- все правила из P и ни одна из цепочек β_i не начинается с A.

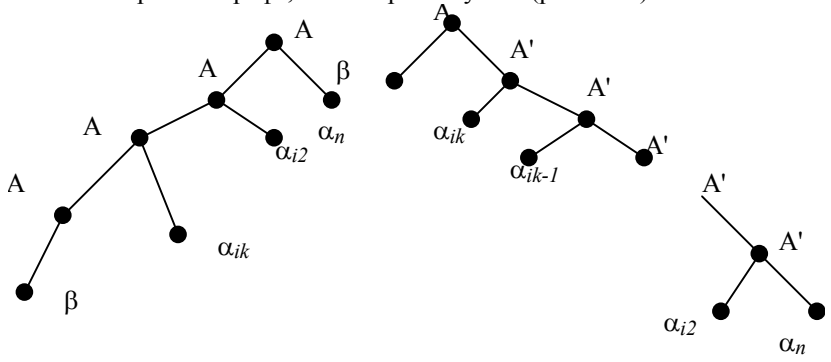
Пусть G'=(N∪{A'}, Σ, P′, S), где A' – новый терминал, а P′ получено из P заменой A – правил правилами

$$A \to \beta_1 \left| \beta_2 \right| \ldots \beta_n \left| \beta_1 A' \right| \beta_2 A' \left| \ldots \beta_n A' \right.$$

$$A' \to \alpha_1 \left| \alpha_2 \right| \cdots \alpha_m \left| \alpha_1 A' \right| \alpha_2 A' \left| \ldots \alpha_m A' \right. .$$

Тогда $L(G')$=$L(G)$.

Рассмотрим на графе, как это реализуется (рис. 3.10).



а) б)

Рис 3.10. Праволинейная а) и леволинейная б) грамматики

Пример:

Пусть G_0 наша обычная грамматика с правилами:

E→E+T | T

T→T*F | F

F→(E) | *a*.

Применяя к ней вышерассмотренную лемму, получаем эквивалентную грамматику с правилами:

E→ T | TE'

$$E' \to +T \mid +TE'$$
$$T \to F \mid FT'$$
$$T' \to *F \mid *FT'$$
$$F \to (E) \mid a.$$

На основании вышеописанного построим алгоритм устранения левой рекурсии. Он будет подобен алгоритму решения уравнения с регулярными коэффициентами.

Алгоритм устранения левой рекурсии.

Вход. Приведённая КС–грамматика $G = (N, \Sigma, P, S)$.

Выход. Эквивалентная КС-грамматика без левой рекурсии.

Метод.

1) Пусть $N = \{A_1 \dots A_n\}$. Преобразуем G так, чтобы в правиле $A_i \to \alpha$ цепочка α начиналась либо с терминала, либо с такого A_i, что $i>j$. С этой целью положим $i=1$.
2) Пусть множество $A_i \to A_i\alpha_i \mid \dots A_i\alpha_m \mid \beta_1 \mid \dots \beta_p$, где ни одна из цепочек β_i не начинается с A_k, если $k \le i$ заменим A_i – правила правилами:
$$A_i \to \beta_1 \mid \dots \beta_p \mid \beta_1 A_i' \mid \dots \beta_p A_i'$$
$$A_i' \to \alpha_1 \mid \dots \alpha_n \mid \alpha_1 A_i' \mid \dots \alpha_1 A_i',$$
где A_i' – новый нетерминал. Правые части всех A_i – правил начинаются теперь с терминала или с A_k, для которого $k>i$.
3) Если $i=n$, то полученную грамматику считаем результатом и останавливаемся, в противном случае $i=i+1$ и $j=1$.
4) Заменим каждое правило $A_i \to A_j\alpha$ правилами $A_i \to \beta_1\alpha \mid \dots \beta_m\alpha$, где $A_j \to \beta_1 \mid \dots \beta_m$ для всех A_j - правил. Так как правая часть каждого A_j правила начинается уже с терминала или A_k для $к>j$, то правая часть каждого A_i - правила будет теперь обладать этими свойствами.
5) Если $j=i-1$, перейти к шагу 2), в противном случае положить $i=i+1$ и перейти к шагу 4).

На основании вышесказанного можно однозначно доказать теорему, что каждый КС-язык определяется нелеворекурсивной грамматикой.

Определение.

КС-грамматика G = (N, Σ, P, S) называется грамматикой в форме Грейбах, если в ней нет *e*- правил и каждое правило из P, отличное от $S \to e$, имеет вид $A \to a\alpha$, где $a \in \Sigma$, $\alpha \in N^*$.

3.11. Автоматы с магазинной памятью

Автоматы с магазинной памятью являются естественной моделью синтаксического анализатора КС-языков.

Автомат с магазинной памятью – это односторонний распознаватель, в потенциально бесконечной памяти которого элементы информации хранятся и используются так же, как и патроны автоматического оружия, т.е. в каждый момент доступен только верхний элемент магазина (рис. 3.11).

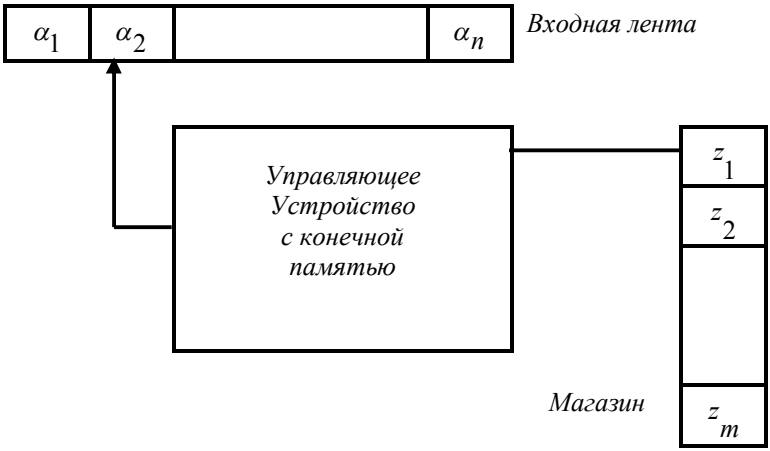


Рис. 3.11. Автомат с магазинной памятью

Все КС-языки определяются недетерминированными автоматами с магазинной памятью, а практически все языки программирования определяются детерминированными автоматами с магазинной памятью.

3.11.1. Основные определения

Определение.

Автомат с магазинной памятью (МП-автомат) – это семерка

$$P = \left(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\right),$$

где Q - конечное множество символов состояния, представляющих всевозможные состояния управляющего устройства;

Σ - конечный входной алфавит;

Γ - конечный алфавит магазинных символов;

δ - отображение множества $Q \times (\Sigma \cup \{e\}) \times \Gamma$ во множество конечных подмножеств множества $Q \times \Sigma^*$;

$q_0 \in Q$ - начальное состояние управляющего устройства;

$Z_0 \in \Gamma$ - символ, находящийся в магазине в начальный момент (начальный символ);

$F \subseteq Q$ - множество заключительных состояний.

Конфигурацией МП-автомата P называется тройка содержащая $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$,

где q - текущее состояние устройства;

w - неиспользованная часть входной цепочки; первый символ цепочки w находится под входной головкой; если $w = e$, то считается, что вся входная лента прочитана;

α - содержимое магазина; самый левый символ цепочки α считается верхним символом магазина; если $\alpha = e$, то магазин считается пустым.

Такт работы МП-автомата P будет представляться в виде бинарного отношения $\vdash$, определенного на конфигурациях. Будем писать

$$(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha),$$

если множество $\delta(q, a, Z)$ содержит (q', γ), где $q, q' \in Q$, $\alpha \in \Sigma \cup \{e\}$, $w \in \Sigma^*$, $Z \in \Gamma$ и $\alpha, \gamma \in \Gamma^*$.

Если $a=e$, то говорят о том, что МП-автомат P, находясь в состоянии q и имея a в качестве текущего входного символа, расположенного под входной головкой, а Z – в качестве верхнего символа магазина, может перейти в состояние q', сдвинуть головку на одну ячейку вправо и заменить верхний символ магазина цепочкой γ магазинных символов. Если $\gamma=e$, то верхний символ удаляется из магазина, тем самым магазинный список сокращается.

Если $a=e$, будем называть этот такт e-тактом. В e-такте текущий входной символ не принимается во внимание и входная головка не сдвигается. Однако состояние управляющего устройства и содержимое

памяти могут измениться. Заметим, что *e*-такт может происходить тогда, когда вся цепочка прочитана.

Начальной конфигурацией МП-автомата *P* называется конфигурация вида (q_0, w, Z_0), где $w \in \Sigma^*$, т.е. управляющее устройство находится в начальном состоянии, входная лента содержит цепочку, которую нужно распознать, и в магазине есть только начальный символ Z_0.

Заключительная конфигурация – это конфигурация вида (q, e, α), где $q \in F$ и $\alpha \in \Gamma^*$.

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

L(*P*) – *язык, определяемый автоматом P* – это множество цепочек, допускаемых автоматом *P*.

Основное свойство МП-автоматов можно сформулировать следующим образом: «То, что происходит с верхним символом магазина, не зависит от того, что находится в магазине под ним».

3.11.2. Эквивалентность МП-автоматов и КС-грамматик

В теории перевода можно показать, что языки определяемые МП-автоматами, – это в точности КС-языки. Начнем с построения естественного (недетерминированного) «нисходящего» распознавателя, эквивалентного данной КС-грамматике.

Лемма.

Пусть $G = (N, \Sigma, P, S)$ - КС-грамматика. По грамматике *G* можно построить такой МП-автомат *R*, что $L_e(R) = L(G)$.

Доказательство.

Построим *R* так, чтобы он моделировал все левые выводы в *G*.

Пусть $R = (\{q\}, \Sigma, N \cup \Sigma, \delta, q, S, \varnothing)$, где δ определяется следующим образом:

1) если $A \to a$ принадлежит *P*, то $\delta(q, e, A)$ содержит (q, α);

2) $\delta(q, a, a) = \{(q, e)\}$ для всех $a \in \Sigma$.

Мы хотим показать, что $A \Rightarrow^m w$ тогда и только тогда, когда $(q, w, A) \vdash^n (q, e, e)$ для некоторых $n, m \geq 1$.

Необходимость этого условия докажем индукцией по *m*. Допустим, что $A \Rightarrow^m w$. Если *m*=1 и $w = a_1 a_2 \dots a_k (k \geq 0)$, то

$$(q, a_1 \dots a_k, A) \vdash (q, a_1 \dots a_k, a_1 \dots a_k) \vdash^k (q, e, e).$$

Теперь предположим, что $A \Rightarrow^m w$ для некоторого *m*>1. Первый шаг этого вывода должен иметь вид $A \Rightarrow X_1 X_2 \dots X_k$, где $X_i \Rightarrow^{m_i} x_i$ для некоторого $m_i < m, 1 \leq i \leq k$ и $x_1 x_2 \dots x_k = w$. Тогда $(q, w, A) \vdash (q, w, X_1 X_2 \dots X_k)$. Если $X_i \in N$, то по предложению индукции $(q, x_i, X_i) \vdash^* (q, e, e)$.

Если $X_i = x_i \in N$, то $(q, x_i, X_i) \vdash (q, e, e)$. Объединяя вместе эти последовательности тактов, видим, что $(q, w, A) \vdash^+ (q, e, e)$.

Для доказательства достаточности покажем индукцией по *n*, что, если $(q, w, A) \vdash^n (q, e, e)$, то $A \Rightarrow^+ w$.

Если *n*=1, то *w*=*e* и $A \to e$ принадлежит *P*. Предположим, что утверждение верно для всех $n' < n$. Тогда первый такт, сделанный МП-автоматом *R*, должен иметь вид $(q, w, A) \vdash (q, w, X_1 X_2 \dots X_k)$, причем $(q, x_i, X_i) \vdash^n_i (q, e, e)$ для $1 \leq i \leq k$ и $x_1 x_2 \dots x_k = w$. Тогда $A \Rightarrow X_1 X_2 \dots X_k$ - правило из *P*, и по предложению индукции $X_i \Rightarrow^+ x_i$ для $X_i \in N$. Если $X_i \in \Sigma$, то $X_i \Rightarrow^0 x_i$. Таким образом

$$\begin{aligned} A &\Rightarrow X_1 \dots X_k \\ &\Rightarrow^* x_1 X_2 \dots X_k \\ &\vdots \\ &\Rightarrow^* x_1 x_2 \dots x_{k-1} X_k \\ &\Rightarrow^* x_1 x_2 \dots x_{k-1} x_k = w \end{aligned}$$

- вывод цепочки *w* из *A* в грамматике *G*.

Контрольные вопросы

1. Способы определения языков.

2. Грамматики.
3. Грамматики с ограничениями на правила.
4. Распознаватели.
5. Регулярные множества, их распознавание и порождения.
6. Алгоритм решения системы линейных выражений с регулярными выражениями.
7. Регулярные множества и конечные автоматы.
8. Проблема разрешимости.
9. Графическое представление конечных автоматов.
10. Минимизация конечных автоматов.
11. Алгоритм построения канонического конечного автомата. Контекстно-свободные грамматики.
12. Деревья выводов. Преобразование КС-грамматик.
13. Алгоритм устранения недостижимых символов.
14. Алгоритм устранения бесполезных символов.
15. Алгоритм преобразования в грамматику без *e*-правил. Алгоритм устранения цепных правил.
16. Грамматики без циклов. Нормальная форма Хомского. Алгоритм преобразования к нормальной форме Хомского.
17. Нормальная форма Грейбах.
18. Алгоритм устранения левой рекурсии.
19. Автоматы с магазинной памятью.

4. КС-грамматики и синтаксический анализ сверху вниз

В практических приложениях нас больше будут интересовать детерминированные МП-автоматы, т. е. такие, которые в каждой конфигурации могут сделать не более одного очередного такта. Языки, определяемые детерминированными МП-автоматами, называются детерминированными КС-языками, а их грамматики — LR (k)-грамматиками.

Определение.

МП-автомат $P = \left(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\right)$ называется *детерминированным* (ДМП), если для каждых $q \in Q$ и $Z \in \Gamma$ либо

1) $\delta\left(q, a, Z\right)$ содержит не более одного элемента для каждого $a \in \Sigma$ и $\delta\left(q, e, Z\right) = \varnothing$, либо
2) $\delta\left(q, a, Z\right) = \varnothing$ для всех $a \in \Sigma$ и $\delta\left(q, e, Z\right)$ содержит не более одного элемента.

Соглашение. Так как ДМП-автомат содержит не более одного элемента, мы будем писать $\delta(q, a, Z) = (r, \gamma)$ вместо $\delta(q, a, Z) = \{(r, \gamma)\}$.

Как уже отмечалось, однотактовые детерминированные МП-автоматы порождают КС-языки, которые называются LR(k)-грамматиками. Те в свою очередь являются частным случаем *s*-грамматик.

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

1) правые части каждого порождающего правила начинаются с *терминала*;
2) в тех случаях, когда в левой части более чем одного порождающего правила появляется *нетерминал*, соответствующие правые части начинаются с различных терминалов.

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

Второе условие соответствует существованию детерминированного одношагового МП-автомата.

Пример.

$S \to pX$ $\qquad$ $X \to x$

$S \to qY$ $\qquad$ $Y \to aYd$

$X \to aXb$ $\qquad$ $Y \to y$

Рассмотрим проблему разбора строки *paaaxbbb* с помощью заданной *s*-грамматики. Начав с символа *S,* попытаемся генерировать строку, применяя левосторонний вывод. Результаты приведены в табл. 4.1.

Таблица 4.1.

Исходная строка	Вывод
paaaxbbb	*S*
paaaxbbb	*PX*
paaaxbbb	*PaXb*
paaaxbbb	*PaaXbb*
paaaxbbb	*PaaaXbbb*
paaaxbbb	*Paaaxbbb*

4.1. LL(1)-грамматики

Если возможно написать детерминированный анализатор, осуществляющий разбор сверху вниз, для языка, генерируемого *s*-

грамматикой, то такой анализатор принято называть *LL*(1)-грамматикой.

Определение.

Обозначения в написании *LL*(1)-грамматики означают:

L – строки разбираются слева направо;

L – используются самые левые выводы;

1 – варианты порождающего правила выбираются с помощью одного предварительного просмотра символа.

Т.е. грамматику называют *LL*(1)-грамматикой, если для каждого нетерминала, появляющегося в левой части более одного порождающего правила, множество направляющих символов, соответствующих правым частям альтернативных порождающих правил, – непересекающиеся.

Определение.

Множество терминальных символов предшественников определяется следующим образом.

$$a \in S(A) \Leftrightarrow A \to^{+} a\alpha ,$$

где *A* – нетерминал;

α - строка терминалов и/или нетерминалов;

$S(A)$ - множество символов предшественников *A*.

Пример.

$P \to Ac$	$A \to Aa$
$P \to Bd$	$B \to b$
$A \to a$	$B \to bB$

символы *a* и *b* – символы предшественники для *P*.

Определение.

Если *A* – нетерминал, то его *направляющими символами* (*DS*) будут $S(A)$+(*все символы, следующие за A, если A может генерировать пустую строку*).

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

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

где *F*(*P*) есть множество символов, которые могут следовать за *P*.

Пример.

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

$P \to e$ $\qquad\qquad$ $C \to f$

$Q \to qQ$

Эта грамматика дает $S(PQ) = \{p, q\}$, $S(BC) = \{b, d\}$, $DS(A, PQ) = \{p, q, b, d\}$ и $DS(A, BC) = \{b, d\}$.

Из определения LL(1)-грамматики следует, что эти грамматики можно разбирать детерминирован сверху вниз.

Алгоритм. Принадлежит ли данная грамматика LL (1)*-грамматике*?

Вход. Некоторая произвольная грамматика.

Выход. «ДА» если данная грамматика является *LL*(1)-грамматикой, «НЕТ» в - противном случае.

Метод.

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

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

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

1) Каждое порождающее правило, имеющее такой символ в правой части, который не может генерировать пустую цепочку (о чем свидетельствует значение соответствующего элемента массива), исключается из грамматики. В том случае, когда для нетерминала в левой части исключенного правила не существует других порождающих правил, значение элемента массива, соответствующего этому нетерминалу, устанавливается на *NO*.
2) Каждый нетерминал в правой части порождающего правила, который может генерировать пустую строку, стирается из правила. В том случае, когда правая часть правила становится пустой, эле-

менту массива, соответствующему нетерминалу в левой части, присваивается значение *YES*, и все порождающие правила для этого нетерминала исключаются из грамматики.

Этот процесс продолжается до тех пор, пока за полный просмотр грамматики не изменится ни одно из значений элементов массива.

Рассмотрим этот процесс на примере грамматики

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

После первого прохода массив будет таким, как показано в табл.4.2, а грамматика сведется к следующей:

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

Таблица 4.2.

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

U – *UNDECIDED* (нерешенный)

Y – *YES* (да)

N – *NO* (нет)

После второго прохода массив будет таким, как показано в табл.4.3, а грамматика примет вид:

1. $A \to XYZ$

Таблица 4.3.

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

Третий проход завершает заполнение массива (табл. 4.4).

Таблица 4.4.

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

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

$$P \to QR$$

$$Q \rightarrow qR$$

можно заключить, что Q есть непосредственный предшественник P, а q – непосредственный предшественник Q. В матрице предшественников для каждого нетерминала отводится строка, а для каждого терминала и нетерминала – столбец. Если нетерминал A, например, имеет в качестве непосредственных предшественников B и C, то в A-ю строку в B-м и C-м столбцах помещаются единицы (табл. 4.5).

Таблица 4.5.

	A	*B*	*C*	...	*Z*	*a*	*b*	*c*	...	*z*
A		*1*	*1*							
B										
C										
D										
.										
.										
.										
Z										

Там, где правая часть правил начинается с нетерминала, необходимо проверить, может ли данный нетерминал генерировать пустую строку, для чего используется массив пустой строки. Если такая генерация возможна, символ, следующий за нетерминалом, является непосредственным предшественником нетерминала в левой части правила и т.д.

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

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

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

процесс рекомендуется выполнить вновь, чтобы поставить единицу в (i, l)-ю позицию, и повторять его до тех пор, пока не будет таких случаев, когда в (i, j)-й позиции и в (j, k)-ой позиции появляются единицы, а в (i, k)-й позиции – нет.

Приведенный выше алгоритм иллюстрирует множество порождающих правил:

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

из которых легко вывести три единицы в матрицу непосредственных предшественников и путем дедукции еще три единицы – в полную матрицу предшествования в соответствии с тем, что X является непосредственным предшественником A, а a – непосредственным предшественником B, а также A (табл. 4.6).

Таблица 4.6.

	A	*B*	*C*	...	*X*	*Y*	*a*	...
A		*1*			*1*		*1*	
B					*1*		*1*	
C								
.								
.								
.								
X							*1*	
Y								

Этот процесс называется нахождением *транзитивного замыкания,* а сама матрица – матрицей достижимости. В этой матрице единицы соответствуют вершинам, между которыми есть соединительные пути.

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

Рассмотренный выше алгоритм используется в двух целях:

1) для определения правильности (принадлежности) данной грамматики к *LL*(1)-грамматике;
2) для преобразования произвольной грамматики к виду *LL*(1).

LL(1) – грамматика очень удобна для организации процесса семантического разбора.

4.2. LL(1)-таблица разбора

Найдя *LL*(1)-грамматику для языка, можно перейти к следующему этапу – применению найденной грамматики в фазе разбора. Обычно модуль компилятора, занимающийся семантическим разбором, называется *драйвером*. Драйвер указывает на то место в синтаксисе, которое соответствует текущему входному символу. Составной частью драйвера является стек, который служит для запоминания адресов возврата всякий раз, когда он входит в новое порождающее правило, соответствующее какому-нибудь нетерминалу.

Опишем сначала возможный вид таблицы разбора, а затем рассмотрим возможные способы ее оптимизации относительно используемых вычислительных ресурсов.

Таблица разбора в общем виде представляет собой одномерный массив структур вида:

declare 1 TABLE,
2 *terminals* **list**,
2 *jump* **int,**
2 *accept* **bool,**
2 *stack* **bool,**
2 *return* **bool,**
2 *error* **bool;**

где **list**

declare 1 list,
2 *term* **string,**
2 *next* **pointer;**

Кроме того, для работы драйвера нужен стек адресов возврата и указатель стека.

В таблице каждому шагу процесса разбора соответствует один элемент. В процессе разбора осуществляются следующие шаги.

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

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

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

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

Рассмотрим схему построения таблицы разбора и соответствующей программы для следующей грамматики:

(1) $PROGRAM \rightarrow begin\ DECLIST\ semi\ STATLIST\ end$
(2) $DECLIST \rightarrow d\ X$
(3) $X \rightarrow comma\ DECLIST$
(4) $X \rightarrow e$
(5) $STATLIST \rightarrow s\ Y$
(6) $Y \rightarrow comma\ STATLIST$
(7) $Y \rightarrow e$

Сначала представим грамматику в виде схемы (рис. 4.1). В скобках и справа на рисунке указаны номера элементов таблицы разбора.

Таблица разбора, соответствующая этой грамматике может быть представлена в виде табл. 4.6.

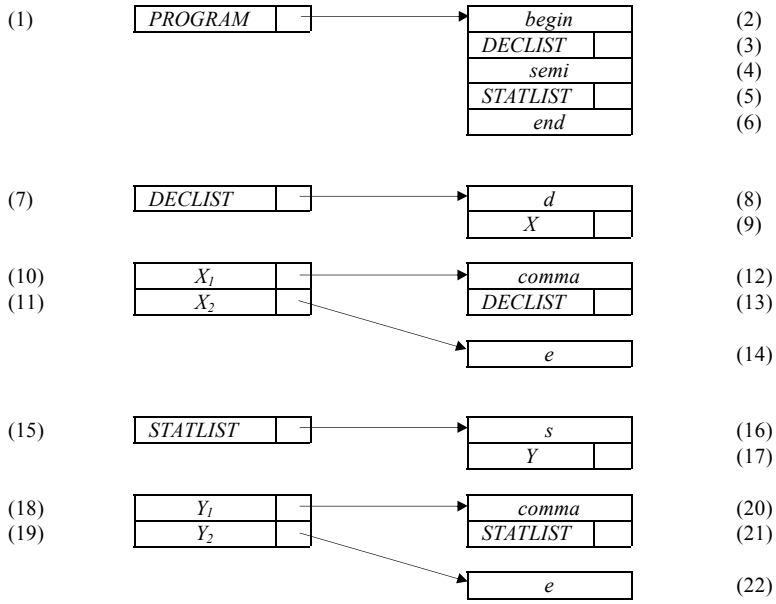


Рис. 4.1. Схема грамматики

Таблица 4.6 – Таблица разбора

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

Рассмотрим разбор предложения

begin d comma d semi s semi end

i	Действия	Стек разбора
1.	*begin* считывается и проверяется; перейти к 2	0
2.	*begin* считывается и принимается; перейти к 3	0
3.	*d* считывается и проверяется; 4 помещается в стек; перейти к 7	4 0
7.	*d* принимается; перейти к 8	4 0
8.	*d* проверяется и принимается; перейти к 9	4 0
9.	*comma* считывается и проверяется; перейти к 10	4 0
10.	*comma* проверяется; перейти к 12	4 0
12.	*comma* проверяется и принимается; перейти к 13	4 0
13.	*d* считывается и проверяется; перейти к 7	4 0
7.	*d* проверяется; перейти к 8	4 0
8.	*d* проверяется и принимается; перейти к 9	4 0
9.	*comma* считывается и проверяется; перейти к 10	4 0
10.	*semi* не совпадает с *comma*; ошибка ложь – перейти к 11	4 0
11.	*comma* проверяется; перейти к 14	4 0
14.	*semi* проверяется; возврат истина – удаляется 4; перейти к 4	0
4.	*semi* проверяется и принимается; перейти к 5	0
5.	*s* считывается и проверяется; 6 помещается в стек; перейти к 15	6 0
15.	*s* проверяется; перейти к 16	6 0
16.	*s* проверяется и принимается; перейти к 17	6 0
17.	*comma* считывается и проверяется; перейти к 18	6 0
18.	*comma* проверяется; перейти к 20	6 0
20.	*comma* проверяется и принимается; перейти к 21	6 0

21.	*s* считывается и проверяется; перейти к 15	6 0
15.	*s* проверяется; перейти к 16	6 0
16.	*s* проверяется и принимается; перейти к 17	6 0
17.	*end* считывается и проверяется; перейти к 18	6 0
18.	*end* не совпадает с *comma*; ошибка ложь; перейти к 19	6 0
19.	*end* проверяется; перейти к 22	6 0
22.	*end* проверяется; возврат истина – удалить 6; перейти к 6	0
6.	*end* проверяется и принимается; возврат истина – удалить 0; *i:=0*	
0.	Разбор заканчивается	

LL(1) – метод разбора имеет ряд преимуществ.

1) Никогда не требует возврата, поскольку этот метод детерминирован.
2) Время разбора пропорционально длине программы.
3) Имеются хорошие диагностические характеристики, и существует возможность исправления ошибок, т.к. синтаксические ошибки распознаются по первому неприемлемому символу, а в таблице разбора есть список возможных символов предложения.
4) Таблица разбора меньше, чем соответствующие таблицы в других методах разбора.
5) LL(1) –разбор применим к широкому классу современных языков.

Контрольные вопросы

1. Эквивалентность МП-автоматов и КС-грамматик.
2. LR(k)-грамматики.
3. S-грамматика.
4. LL(1)-грамматика.
5. Алгоритм определения принадлежности данной грамматики к LL(1)-грамматике.
6. LL(1) – таблица разбора.

5. Синтаксический анализ снизу вверх

5.1. Разбор снизу вверх

Мы рассмотрели проблему левостороннего разбора как частный случай синтаксического анализа. Обобщая выводы, можно констатировать, что методы разбора могут быть нисходящими, т.е. идущими от стартового символа к предложению, и восходящими - от предложения к стартовому символу. Предложения, читаемые слева направо, норма для большинства естественных языков, хотя проблема разбора в них не всегда тривиальна. Однако ряд естественных языков и языков программирования имеют другую структуру. В этом случае удобнее использовать правосторонний вывод и соответствующие грамматики. Эти грамматики называются LR-грамматиками и в синтаксическом разборе используют технологию снизу вверх.

Синтаксические анализаторы, работающие по этому принципу, сводят предложения языка к начальному символу путем последовательного применения правил грамматики.

Рассмотрим, например, язык, генерируемый правилами

1) $S \rightarrow$ **real** *IDLIST*
2) *IDLIST* $\rightarrow$ *IDLIST, ID*
3) *IDLIST* $\rightarrow$ *ID*
4) *ID* $\rightarrow A \mid B \mid C \mid D$.

Предложение **real** *A, B, C* принадлежит этому языку и может быть разобрано по следующей схеме. Каждый символ, считанный анализатором, немедленно помещается в стек анализатора.

real $\uparrow$*A, B, C*

real *A*$\uparrow$, *B, C*

Стрелка ставится непосредственно после последнего считанного символа. Затем анализатор заменяет *A* с помощью правила 4). Это действие называется *приведение*. Правые части правил заменяются их соответствующими левыми частями.

real *A*$\uparrow$, *B, C*

Далее применяется правило (3) для выполнения другого приведения.

real *A*↑, *B, C* *IDLIST*
real

Теперь анализатор считывает следующий символ

,
real *A,* ↑*B, C* *IDLIST*
real
B

и другой

,
real *A, B*↑, *C* *IDLIST*
real
ID

Очередные два действия – приведение с использованием правил 4) и 2)

,
real *A, B*↑, *C* *IDLIST*
real

real *A, B*↑, *C* *IDLIS*
real

и последующим считыванием еще двух символов

,
real *A, B,* ↑ *C* *IDLIST*
real
C

,
real *A, B, C*↑ *IDLIST*
real
ID

Последние три действия – это приведение с использованием правил 4), 2) и 1).

,
real *A, B, C*↑ *IDLIST*
real

real *A, B, C*↑ *IDLIST*
real

real *A, B, C*↑ *S*

Разбор считается завершенным, когда в стеке останется только начальный символ и предложение считано целиком. Стек разбора соответствует части автомата с магазинной памятью.

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

1) *сдвиг*, во время которого считывается и помещается в стек символ. Это соответствует движению на один пункт вдоль какого-нибудь правила грамматики;
2) *приведение*, во время которого множество элементов в верхней части стека замещаются каким-либо нетерминалом грамматики с помощью одного из порождающих правил этой грамматики.

5.2. LR(1) - таблица разбора

Выше мы рассмотрели технологию разбора «снизу вверх». Однако при этом не обсуждался вопрос, как решить, следует ли выполнять конкретное приведение, когда правая часть правила появляется в вершине стека. Механизмом, который регулирует вопрос принятия правильного решения, является таблица разбора.

Таблица представляет собой матрицу. Она состоит из столбцов для каждого терминала и нетерминала грамматики плюс знак окончания и строк, соответствующих каждому состоянию, в котором может находиться анализатор. Каждое состояние соответствует той позиции в порождающем правиле, которой достиг анализатор. Таблица разбора для грамматики

1) $S \rightarrow$ **real** *IDLIST*
2) $IDLIST \rightarrow IDLIST, ID$
3) $IDLIST \rightarrow ID$
4) $ID \rightarrow A \mid B \mid C \mid D$

приведена в табл. 5.1. Драйвер анализатора использует еще и два стека – стек *символов* и стек *состояний*. Таблица разбора включает элементы четырех типов.

1) *Элементы сдвига*. Эти элементы имеют вид *S7* и означают: поместить в стек символ, соответствующий столбцу; поместить в стек состояний 7 и перейти к состоянию 7; если входной символ – терминал, принять его.
2) *Элемент приведения*. Он имеет вид *R4* и означает: выполнить приведение с помощью правила 4), т.е., допустив, что *n* есть число символов в правой части правила 4), удалить *n* элементов из стека символов и *n* элементов из стека состояний и перейти к состоянию, указанному в верхней части стека состояний. Нетерминал в левой части правила (4) нужно считать следующим входным символом.
3) *Элемент ошибок*. Эти элементы являются пробелами в таблице и соответствуют синтаксическим ошибкам.
4) *Элемент остановки*. Им завершается разбор.

Таблица 5.1.

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

Рассмотрим, как с помощью вышеописанной таблицы разбирается строка:

real *A, B, C* ⊥

	Стек символов	Стек состояний
↑**real** *A, B, C* ⊥		1

входной символ **real** – из элемента таблицы (1, **real**), сдвиг в состояние 2;

real↑ *A, B, C* ⊥	**real**	2 1

входной символ *A* – сдвиг в состояние 3;

real *A*↑, *B, C* ⊥	*A* **real**	3 2 1

входной символ – «,», приведение по правилу (4);

real *A*↑, *B, C* ⊥	**real**	2 1

входной символ – *ID*, сдвиг в состояние 4;

real *A*↑, *B, C* ⊥	*ID* **real**	4 2 1

входной символ – «,», приведение по правилу (3);

real *A*↑, *B, C* ⊥	**real**	2 1

входной символ – *IDLIST*, сдвиг в состояние 5;

real *A*↑, *B, C* ⊥	*IDLIST* **real**	5 2 1

входной символ – «,», сдвиг в состояние 6;

real *A*↑, *B*, *C* ⊥

,	6
IDLIST	5
real	2
	1

входной символ – *B*, сдвиг в состояние 3;

real *A*, *B*↑, *C* ⊥

B	3
,	6
IDLIST	5
real	2
	1

входной символ – «,», приведение по правилу (4);

real *A*, *B*↑, *C* ⊥

,	6
IDLIST	5
real	2
	1

входной символ – *ID*, сдвиг в состояние 7;

real *A*, *B*↑, *C* ⊥

ID	7
,	6
IDLIST	5
real	2
	1

входной символ – «,», приведение по правилу (2);

real *A*, *B*↑, *C* ⊥

real	2
	1

входной символ – *IDLIST*, сдвиг в состояние 5;

real *A*, *B*↑, *C* ⊥

IDLIST	5
real	2
	1

входной символ – «,», сдвиг в состояние 6;

real *A*, *B*, ↑*C* ⊥

,	6
IDLIST	5
real	2
	1

входной символ – *C*, сдвиг в состояние 3;

real *A*, *B*, *C*↑ ⊥

C	3
,	6
IDLIST	5
real	2
	1

входной символ – «⊥», приведение по правилу (4);

real *A, B, C*↑ ⊥	,	6
	IDLIST	5
	real	2
		1

входной символ – *ID*, сдвиг в состояние 7;

	ID	7
	,	6
real *A, B, C*↑ ⊥	*IDLIST*	5
	real	2
		1

входной символ – «⊥», приведение по правилу (2);

real *A, B, C*↑ ⊥	**real**	2
		1

входной символ – *IDLIST*, сдвиг в состояние 5;

		5
real *A, B, C*↑ ⊥	*IDLIST*	2
	real	1

входной символ – «⊥», приведение по правилу (1);

real *A, B, C*↑ ⊥		1

входной символ – *S*, поэтому *HALT* (*ОСТАНОВ*).

Разбор завершен успешно.

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

5.3. Построение LR – таблицы разбора

При построении LR-таблиц разбора нам необходимо ссылаться на конкретную позицию в правиле, поэтому в правилах вводится понятие «*конфигурация*». Например, в грамматике

1) $S \to$ **.** **real** *IDLIST*
2) $IDLIST \to IDLIST,\ ID$
3) $IDLIST \to ID$
4) $ID \to A \mid B \mid C \mid D$

точка соответствует конфигурации (1,0), т.е. правило 1) позиция 0; конфигурация (1,1) соответствует точке, появляющейся сразу после **real** в правиле 1), а (2,0) – точке появляющейся перед *IDLIST* в правой части правила 2). Так, конфигурация (2,2) показывает, что правая часть правила 2) распознана по запятую включительно.

Состояние в таблице разбора примерно соответствует конфигурациям в грамматике с той разницей, что конфигурации, которые неразличимы для анализатора, представляются одним и тем же состоянием. Например, если (1,0) соответствует состоянию 1, а (1,1) - состоянию 2, то в вышеприведенной грамматике (2,0), (3,0) и (4,0) будут также соответствовать состоянию 2. В этом случае говорят, что множество конфигураций

$$\{(1,1),(2,0),(3,0),(4,0)\}$$

образуют замыкание (1,1).

Из заданного состояния, не соответствующего *концу правила*, можно перейти в другое состояние, введя терминальный или нетерминальный символ. Это состояние называется *преемником* первоначального состояния. Чтобы построить таблицу разбора, необходимо прежде найти все состояния в грамматике. Поэтому, начиная с конфигурации (1,0), последовательно выполним операции замыкания и образования приемника до тех пор, пока все конфигурации не окажутся включенными в какое-либо состояние. Там, где ряд конфигураций содержится в одном замыкании, каждая из них будет соответствовать одному и тому же состоянию. Новая конфигурация, которая получается при операции образования преемника, называется *базовой*. Если за базовой конфигурацией следует нетерминал, то все конфигурации, соответствующие помещению точки слева от каждой правой части для данного нетерминала, сконцентрируются в замыкании этой базовой конфигурации. В приведенной грамматике можно выделить семь состояний, которые описываются следующим образом (табл. 5.2).

Таблица 5.2 – Состояния грамматики

Состояние	База	Замыкание
1	(1,0)	{(1,0)}
2	(1,1)	{(1,1), (2,0), (3,0), (4,0)}
3	(4,1)	{(4,1)}
4	(3,1)	{(3,1)}
5	{(2,1), (1,2)}	{(2,1), (1,2)}
6	(2,2)	{(2,2), (4,0)}
7	(2,3)	{(2,3)}

Эти состояния расположены в грамматике следующим образом:

1) $S \rightarrow_1$ **real** $_2\ IDLIST_5$
2) $IDLIST \rightarrow_2 IDLIST_5\ ,_6\ ID_7$
3) $IDLIST \rightarrow_2 ID_4$
4) $ID \rightarrow_{(2,6)} A \mid B \mid C \mid D_3$

Заметим, что конфигурация может соответствовать более чем одному состоянию, и в базе может быть более одной конфигурации, если преемники двух конфигураций в одном и том же замыкании неразличимы. Например, за конфигурациями (1,1) и (2,0) следует *IDLIST*, что делает (1,2) и (2,1) неразличимыми, пока не осуществится операция замыкания. Число состояний в анализаторе соответствует числу множеств неразличимых конфигураций в грамматике. Причина того, что два и более состояния соответствуют одной конфигурации, раскрывается в ходе разбора.

Действия анализатора со сдвигом аналогичны операции получения преемника. Поэтому действия со сдвигом в таблицу разбора могут вноситься на основании информации о размещении состояний в грамматике (табл. 5.3).

Таблица 5.3.

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

Например, правило 2) означает «из состояния 2 при чтении *IDLIST* перейти в состояние 5», «из состояния 5 при чтении *запятой* перейти в состояние 6» и т.д.

Задача внесения приведения в таблицу разбора нетривиальна. Однако, единственные состояния, в которых приведения возможны, - это состояния, соответствующие окончаниям правил (в нашей грамматике 3, 4, 5, и 7). Поэтому мы можем внести *R4* во все столбцы состояния 3, *R3* - во все столбцы состояния 4, *R1* – во все столбцы состояния 5 и *R2* – во все столбцы состояния 7. Однако в состоянии 5 в одном столбце уже имеется элемент сдвига. Таким образом, возникает конфликт *сдвиг/приведение.* Состояние 5 называется *неадекватным*. Разрешить эту проблему можно просматривая символы, которые показали бы приведение в состояние 5, а не сдвиг. Из правил 1) и 2) следует, что такими символами могут быть только «⊥» и «,», а приведение возможно лишь в том случае, если символ окажется «⊥», в то время как анализатор осуществит сдвиг в состояние 6, если следующим символом

будет «,». Поэтому вносим *R1* в пятую строку столбца, соответствующего знаку «⊥». Этим действием неадекватность снимается (табл. 5.4).

Таблица 5.4.

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

Аналогичным образом, рассмотрев предварительные символы, можно исключить из таблицы все лишние приведения. В результате таблица разбора приобретает вид табл. 5.5.

Таблица 5.5.

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

5.4. Сравнение LL – и LR – методов разбора

Как LL - , так и LR – методы разбора имеют много достоинств. Оба метода – детерминированы и могут обнаруживать синтаксические ошибки на самом раннем этапе. LR – методы обладают тем преимуществом, что они применимы к более широкому классу грамматик и языков и преобразование грамматик в них обычно не требуется. Однако, для хорошо структурированной грамматики сам факт преобразования грамматики не вызывает каких-либо технических трудностей.

Два этих метода можно сравнивать в отношении размеров таблиц разбора и времени разбора. Использование по одному слову на элемент в LL – таблице разбора позволяет свести размер типичной таблицы к минимуму, в то время как LR – разбор этой возможности не предоставляет.

Коэн и Рот сравнивали максимальное и среднее время разбора с помощью LL – и LR – анализаторов. Из данных для машин серии DEC результаты разбора LL – методом оказались быстрее на 50%.

Оба метода разбора позволяют включать в синтаксис действия для выполнения некоторых аспектов компиляции. Для LR – анализаторов такие действия обычно связаны с приведениями.

Разные разработчики компиляторов отдают предпочтение разным методам (т.е. разбору снизу вверх или сверху вниз), что постоянно служит предметом дискуссий. На практике выбор метода в основном зависит от наличия хорошего генератора синтаксических анализаторов любого типа.

Контрольные вопросы

1. Технология разбора снизу вверх.
2. Построение LR – таблицы разбора.
3. Метод определения количества состояний грамматики.
4. Разрешение конфликта сдвиг/приведение.
5. Сравнение LL - и LR – методов разбора.

6. Оптимизация кода

Одной из трудных и неопределенных проблем, возникающих при построение компиляторов, является генерация хорошего объектного кода. Качество программы определяется по времени её выполнения и размерам. К сожалению, для данной конкретной программы невозможно установить время выполнения самой быстрой эквивалентной программы или длину самой короткой эквивалентной программы.

Большинство алгоритмов улучшения кода можно рассматривать, как приложение различных преобразований к некоторому промежуточному представлению исходной программы с целью привести её к форме из которой можно получить наиболее эффективный объектный код. Эти преобразования могут применяться в любой точке процесса компиляции. Однако, наиболее часто эти преобразования применяются к программе на промежуточном языке.

Преобразования по улучшению кода можно разделить на машинно-независимые и машинно-зависимые. Примером машинно-независимой оптимизации может служить удаление из программы бесполезных операций, т.е. таких, которые никаким образом не влияют на выход из программы.

С помощью машинно-зависимых преобразований можно попытаться привести программу к форме, в которой могут сказываться преимущества, связанные с машинными командами специального вида. Эти виды преобразований мы рассматривать не будем.

6.1. Оптимизация линейного участка

Рассмотрим схему программы, представляющей собой последовательность операторов присвоения, каждый из которых имеет вид $A \leftarrow f(B_1, B_2, \ldots B_k)$, где A, и B_i - скалярные переменные, а f – функция от r переменных.

6.1.1. Модель линейного участка

Начнем с определения *блока***.** Блок моделирует часть программы на промежуточном языке, которая содержит только операторы присвоения.

Определение. Пусть $\sum$ - счетное множество *имен переменных*, а θ-конечное множество *операций*. Предположим, что каждая операция $\theta \in \Theta$ имеет известное фиксированное количество операторов. Θ и $\sum$ не пересекаются.

Оператором называется цепочка вида

$A \rightarrow \theta\ B_1, B_2, \ldots B_r$

где $A, B_1, B_2, \ldots B_r$ переменные из $\sum$, а θ - это r – местная операция из Θ. Будем говорить, что оператор осуществляет присвоения переменной A и ссылается $B_1, B_2, \ldots B_r$.

Блок $\mathcal{B}$ – это тройка (P, I, U), где

1) P – список операторов $S_1, S_2, \ldots S_n$ $(n \geq 0)$,
2) I – множество входных переменных,
3) U – множество выходных переменных.

Будем предполагать, что если оператор S_1 ссылается на A, то A - либо входная переменная, либо осуществлено присвоение ей значения некоторым оператором до S; (т.е. некоторым S_i $I < j$). Таким образом, внутри блока все переменные, на которые ссылается, к этому моменту

определены либо внутренним образом, как переменные, которым присвоены значения, либо внешним как входные переменные.

Аналогичным образом будем предполагать, что каждая выходная переменная либо является входной переменной, либо ей присвоено значение некоторым оператором.

Пример.

$A \leftarrow + B\ C$, перфиксная запись оператора $A \leftarrow B+C$.

Если оператор осуществляет присвоения, то с этим присвоением можно связать некоторое «значение». Последнее представляет собой формулу для A в терминах первоначальных значений входных переменных.

Будем предполагать, что входные переменные имеют неизвестные значения и их надо рассматривать как алгебраически неизвестные. Кроме того, значения операций и множество, на котором они определенны, не конкретизированы, так что в качестве значений мы можем рассматривать лишь формулу, а не некоторую величину.

Определение. Пусть (P, I, U) – блок, $P = S_1; S_2;\ldots; S_n$. Определим значение $v_t(A)$ переменной A непосредственно после момента времени $0 \le t \le n$, как префиксное выражение:

1) Если $A \in I$, то $v_0(A) = A$.

2) Если оператором S_t является $A \leftarrow \theta B_1 \ldots B_r$, то

a) $v_t(A) = \theta v_{t-1}(B_1) \ldots v_{t-1}(B_r)$,

b) $v_t(C) = v_{t-1}(C)$ для всех $C \neq A$, при условии, что $v_t(C)$ определено.

3) Для всех $A \in \Sigma$, если $v_t(A)$ не определенно по 1) или 2), то оно неопределенно.

Два блока считаются эквивалентными ($\equiv$) если они имеют одно и тоже значение, т.е. цепочки образующие префиксные выражения, равны тогда и только тогда, когда они тождественны.

Пример. Пусть $I = \{A, B\}$, $U = \{F, G\}$ и P состоит из операторов

$T \leftarrow A+B$

$S \leftarrow A-B$

$T \leftarrow T*T$

$S \leftarrow S*S$

$F \leftarrow T+S$

$G \leftarrow T-S$.

Сначала $v_0(A) = A$ и $v_0(B) = B$. После первого операнда $v_1(B) = B$. После второго оператора $v_2(S) = A-B$, а значение остальных прежнее. После третьего оператора $v_3(T) = (A+B)*(A+B)$, остальные значения переносятся с предыдущего шага:

$v_4(S) = (A-B)*(A-B)$

$v_5(F) = (A+B)*(A+B)+(A-B)*(A-B)$

$v_6(G) = (A+B)*(A+B)-(A-B)*(A-B)$

Поскольку $v_6(F) = v_5(F)$, значение блока будет

$\{(A+B)*(A-B)+(A-B)*(A-B), (A+B)*(A+B)-(A-B)*(A-B)\}$.

При оптимизации кода мы не будем пока рассматривать A и B как массивы.

Наши основные предположения:

1) Важным фактором, касающимся блока, является набор функций входных переменных (переменных определенных для блока), вычисляемых внутри блока. Сколько раз вычисляется конкретная функция несущественно.

2) Имена переменных, участвующих в вычислениях, несущественны.

3) Мы не включаем операторы вида $X \leftarrow Y$.

6.1.2. Преобразование блока

Если даны два блока $\mathcal{B}_1$ и $\mathcal{B}_2$, то можно установить, эквивалентны ли они, вычислив их значения и выяснив, выполняются ли равенство $v(\mathcal{B}_1) = v(\mathcal{B}_2)$. Однако можно указать бесконечное число блоков, эквивалентных любому данному.

Например, если $\mathcal{B} = (P, I, U)$ – блок, X- переменных, на которых нет ссылки в $\mathcal{B}$, A- входная переменная, а θ- операция, то к P можно любое количество раз присоединить оператор $X \leftarrow \theta A \ldots A$ не изменяя значение $\mathcal{B}$.

Если выбрать необходимую оценку, то не все эквивалентные блоки будут одинаково эффективными. При данном блоке $\mathcal{B}$ существуют различные преобразования, применяемые для отображения его в эквивалентный и возможно более желательный блок $\mathcal{B}'$. Пусть $\mathcal{F}$ – множество всех таких преобразований, сохраняющих эквивалентность блока. Любое преобразование из $\mathcal{F}$ можно осуществить с помощью конечной последовательности четырех простейших преобразований блоков.

Определение. Пусть $\mathcal{B} = (P, I, U)$ – блок, $P = S_1; S_2; \ldots; S_n$. Для однообразия примем, что все элементы входного множества приписаны к некоторому нулевому оператору S_0, а все элементы выходного множества - к некоторому $n+1$ оператору S_{n+1}.

Переменная A называется активной после момента времени t, если:

1) ей присвоено значение некоторым оператором S_i,

2) ей не присвоено значение операторами S_{i+1}, S_{i+2}, ... , S_j,
3) на нее не ссылается оператор S_{i+1},
4) $0 \le i \le t \le j \le n$.

Если число j имеет максимально возможное значение, то последовательность операторов S_{i+1}, S_{i+2}, ... , S_{j+1} называется областью действия оператора S_i и областью действия этого присвоения переменной A.

Если A входная переменная и после S_i, ей не присваивается значение, то $j=n+1$, и говорят, что U лежит в области действия оператора S_i.

Если блок содержит такой оператор S, что переменная, которой присваивается значение в S, не является активной после этого оператора, то области действия оператора S пуста, и говорят, что *S- бесполезный оператор.* (S - бесполезный оператор, если он не присваивает значение переменной, которая не является выходной и на которую нет ссылки в последующих операторах).

Пример. Рассмотрим блок, в котором α, β и γ списки из нуля или более операторов:

α
$A \leftarrow B+C$
β
$D \leftarrow A*E$
γ.

Если A не присваивает значение в последовательности операторов β и на нее, нет ссылок из γ, то область действия оператора $A \leftarrow B+C$ включает полностью в и оператор $D \leftarrow A*E$. Если в γ нет операторов, ссылающихся на D, и D не является выходной переменной, то оператор $D \leftarrow A*E$ бесполезен.

Определим теперь четыре простейших преобразования блоков, сохраняющих эквивалентность.

Пусть $\mathcal{B} = (P, I, U)$- блок и $P = S_1; S_2;...; S_n$. Преобразования определим в терминах их воздействия на блок.

T_1: *Удаление бесполезных присваиваний*

Если оператор S_i, $0 \le i \le n$ присваивает значение переменной A и она не активна после момента i, то

1) при $I > 0$ можно удалить S_i из P,
2) при $I = 0$ можно удалить A из I.

Пример. Пусть $\mathcal{B} = (P, I, U)$, где $I = \{A, B, C\}$, $U = \{F, G\}$ и P состоит из правил:

$F \leftarrow A+A$

$G \leftarrow F*C$
$F \leftarrow A+B$
$G \leftarrow A*B.$

Второй оператор бесполезен т.к. его область действия пуста. Таким образом одно применение T_1 отображает $\mathcal{B}$ в $\mathcal{B}_1 = (P_1, I_1, U_1)$ где P_1

$F \leftarrow A+A$
$F \leftarrow A+B$
$G \leftarrow A*B.$

Теперь в $\mathcal{B}_1$ бесполезна переменная C и первый оператор. Применяя повторно T_1 можно получить $\mathcal{B}_2 = \{P_2, \{A, B\}, U\}$, где P_2 состоит из

$F \leftarrow A+B$
$G \leftarrow A*B.$

Для того, чтобы можно было систематически удалить из блока $\mathcal{B} = (P, I, U)$ все бесполезные операторы, надо определить множество переменных (тех, которые явно или неявно используются в вычислениях выхода) после каждого оператора блока, начиная с последнего оператора в P и поднимаясь вверх. Ясно, что $U_n = U$ - множество всех переменных, полезных после последнего оператора S_n.

Предположим, что оператором S_i является $A \leftarrow \theta\ B_1B_2$, B_r и U_i – множество переменных полезных после S_i.

1) Если $A \in V_i$, то S_i – полезный оператор, так как переменная A используется для вычисления выходной переменной Тогда множество U_{i-1} полезных переменных после S_{i-1} находится заменой A в U_i на переменные $B_1, B_2, \ldots, B_r$ (т.е. $U_{i-1} = (U_i - \{A\}) \cup \{B_1, B_2, \ldots, B_r\}$).

2) Если $A \notin U$i, то оператор S_i бесполезен, его можно удалить. В этом случае $U_{i-1} = U_i$.

3) После вычисления U_0 можно удалить из I все входные переменные, которых нет в U_0.

T_2: *Исключение избыточных вычислений*

Предположим, что $\mathcal{B} = (P, I, U)$ – блок, где P имеет вид

α
$A \leftarrow \theta C_1 \ldots C_r$
β
$B \leftarrow \theta C_1 \ldots C_r$
γ

причем ни одна из переменных C_1, C_2,…,C_r ни есть A, ни одной из них не присваивается значение в ни в каком операторе β,

Преобразование T_2 отображает $\mathcal{B}$ в $\mathcal{B}' = (P', I, U')$ где P' правила:

α

$D \leftarrow \theta C_1 \ldots C_r$

β′

γ′

и

1) β′- это список β, в котором все ссылки на переменную A в области действия данного изображения этой переменной заменены ссылками на P,

2) γ′ - это список γ, в котором все ссылки на A и B в области данных изображений заменены ссылками на D.

Если область действия переменной A или B в областях действия изображений распространяется на S_{n+1}, по U'- это множество U, в котором A или B заменены на D.

D - может быть любым именем, не меняющим значение блока.

Пример.

$S \leftarrow A+B$

$F \leftarrow A+S$

$R \leftarrow B+B$

$T \leftarrow A*S$

$G \leftarrow T*R$.

Второй и четвертый операторы дают избыточные вычисления, так что к $\mathcal{B}$ можно применить преобразования T_2. В результате чего получим $\mathcal{B}' = (P', \{A, B\}, \{D, G\})$, где P'

$S \leftarrow A+B$

$D \leftarrow A*S$

$R \leftarrow B+B$

$G \leftarrow D*R$.

T_3: *Переименование*

Ясно, что, поскольку речь идет о значение блока $\mathcal{B} = (P, I, U)$, имена переменных, которым присваиваются значения, не существенны. Предположим, что оператором S_i в P является $A \leftarrow \theta B_1 \ldots B_r$ и переменная C не является активна в области действия оператора S_i. Тогда можно положить $\mathcal{B}' = (P', I, U')$ где P' - это P в котором S_i заменен на

$C \leftarrow \theta B_1 \ldots B_r$, а все вхождения A в области действия оператора S_i заменены на C. Если U лежит в области оператора S_i, то U' - это U, в котором переменная A заменена на C. В противном случае $U' = U$.

Пример. Пусть $\mathcal{B} = (P, \{A, B\}, \{F\})$, где P состоит из

$T \leftarrow A*B$

$T \leftarrow T+A$

$F \leftarrow T*T$

Одно применение T_3 позволяет изменить имя переменной T, на S. Таким образом $\mathcal{B}' = (P', \{A, B\}, \{F\})$

$S \leftarrow A*B$

$T \leftarrow S+A$

$F \leftarrow T*T$.

T_4: *Перестановка*

Пусть $\mathcal{B}=(P, I, U)$ - блок, в котором оператором S_i является $A \leftarrow \theta B_1 \ldots B_r$ оператором S_{i+1} является $C \leftarrow \psi D_1 \ldots D_s$, A не совпадает ни с одной переменной $C, D_1, \ldots D_s$ и C не совпадает с и с одной из переменных из $A, B_1 \ldots B_r$ тогда преобразование T_4 отображает блок $\mathcal{B}$ в $\mathcal{B}'=(P', I, U')$ где P' - это P в котором S_i и S_{i+1} переставлены.

Пример. Пусть $\mathcal{B}=(P, \{A, B\}, \{F, G\})$, где P состоят из правил

$F \leftarrow A+B$

$G \leftarrow A*B$.

Можно применить T_4 и отобразить $\mathcal{B}$ в $(P', \{A, B\}, \{F, G\})$, где P' состоит из правил

$G \leftarrow A*B$

$F \leftarrow A+B$.

6.1.3. Графическое представление блоков

Для каждого блока $\mathcal{B}=(P, I, U)$ можно найти ориентированный ациклический граф D, естественным образом представляющий $\mathcal{B}$. Каждый лист графа D соответствует одной входной переменной в I, а каждая его внутренняя переменная вершина- оператор из P. К таким графам легко применить рассмотренные нами преобразования.

Определение. Пусть $\mathcal{B}=(P, I, U)$ – блок. Построим помеченный граф $D(\mathcal{B})$:

1) Пусть $P=S_1, S_2, \ldots, S_n$

2) Для каждой переменной $A \in I$ образуем вершину с меткой A и будем называть её последним определением для A.

3) Для i=1,2…n делаем следующие: пусть $A \leftarrow \theta B_1\ B_2\ \ldots\ B_r$ образуем новую вершину, помеченную θ, из которой выходят r ориентированных дуг. Пусть j дуга (при упорядоченье дуг слева направо) указывает на последнее определение для B_j $1 \le j \le r$. Новая вершина, помечена θ, становится последним определением A этой вершине соответствует оператору S_i в D.

4) После шага 3) вершины, являющиеся последним определением входных переменных, помечаются как выделенные и отмечаются кружками

Пример: пусть $\mathcal{B}=(P, \{A, B\}, \{F, G\})$- блок в котором P составляет из операторов. Граф $D(\mathcal{B})$ изображен на рис. 11.1.

$T \leftarrow A+B$

$F \leftarrow A*T$

$T \leftarrow B+F$

$G \leftarrow B*T$

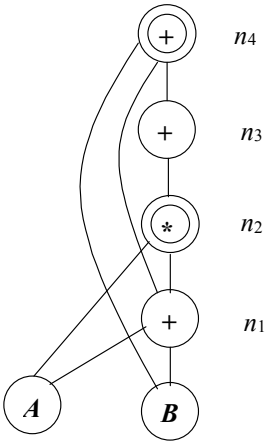


Рис. 11.1. Пример ориентированного ациклического графа.

Четыре оператора из блока $\mathcal{B}$ соответствуют по порядку вершинам n_1, n_2, n_3 и n_4

Каждый граф представляет класс эквивалентности $\overset{*}{\underset{3,4}{\Leftrightarrow}}$. Если блок $\mathcal{B}_1$ с помощью последовательности преобразований T_3 и T_4 можно преобразовать в блок $\mathcal{B}_2$, то блок $\mathcal{B}_1$ и $\mathcal{B}_2$ имеют один и тот же граф и обратно.

Лемма. Если $\mathcal{B}_1 \Rightarrow 3,4\ \mathcal{B}_2$, то $D(\mathcal{B}_1)=D(\mathcal{B}_2)$

Определение. Блок $\mathcal{B} = (P, I, V)$ называется открытым если

1) ни один из операторов P не имеет вид $A \leftarrow \alpha$ где $A \in I$,
2) в P нет двух операторов, присваивающих значение одной и той же переменной.

В открытом блоке $\mathcal{B}=(P, I, U)$ все операторы S_i из P присваивают значения переменным X_i, не входящим в I. Открытый блок всегда можно получить с помощью только преобразований T_3.

Лемма. Пусть $\mathcal{B}=(P, I, U)$- блок. Тогда существует такой эквивалентный открытый блок $\mathcal{B}'=(P',I',U')$, что $\mathcal{B} \Leftrightarrow 3\ \mathcal{B}'$

Теорема. $D(\mathcal{B}_1) = D(\mathcal{B}_2)$ тогда и только тогда, когда $\mathcal{B}_1 \overset{*}{\underset{3,4}{\Leftrightarrow}} \mathcal{B}_2$

Т.е. два блока имеют один и тот же граф тогда и только тогда, когда их можно преобразовать в один в другой переименованием и перестановкой.

Следствие. Если $D(\mathcal{B}_1) = D(\mathcal{B}_2)$, то $\mathcal{B}_1 \equiv \mathcal{B}_2$.

Пример. Рассмотрим два блока $\mathcal{B}_1 = \{P_1, \{A, B\}, \{F\}\}$ и $\mathcal{B}_1 = \{P_2, \{A, B\}, \{F\}\}$, множества P_1 и P_2 для них приведены в табл. 11.1.

Таблица 11.1.

P_1	P_2
$C \leftarrow A*A$	$C \leftarrow B*B$
$D \leftarrow B*B$	$D \leftarrow A*A$
$E \leftarrow C-D$	$E \leftarrow D+C$
$F \leftarrow C+D$	$C \leftarrow D-C$
$\underline{F \leftarrow E/F}$	$C \leftarrow C/E$

Блоки $\mathcal{B}_1$ и $\mathcal{B}_2$ имеют один и тот же граф, изображенный на рис. 11.2.

С помощью преобразования T_3 можно отобразить $\mathcal{B}_1$ и $\mathcal{B}_2$ в открытые блоки $\mathcal{B}'_1=(P'_1,\{A, B\},\{X_5\})$ и $\mathcal{B}'_2=(P'_2,\{A, B\},\{X_5\})$ (табл. 11.2).

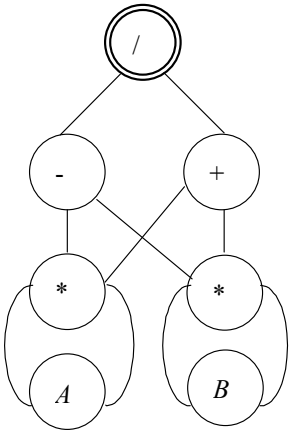


Рис. 11.2. Граф для $\mathcal{B}_1$ и $\mathcal{B}_2$.

Таблица 11.2.

P'_1	P'_2
$X_1 \leftarrow A*A$	$X_2 \leftarrow B*B$
$X_2 \leftarrow B*B$	$X_1 \leftarrow A*A$
$X_3 \leftarrow X_1 - X_2$	$X_4 \leftarrow X_1 + X_2$
$X_4 \leftarrow X_1 + X_2$	$X_3 \leftarrow X_1 - X_2$
$X_5 \leftarrow X_3 / X_4$	$X_5 \leftarrow X_3 / X_4$

А с помощью оператора перестановки привести и сделать полностью эквивалентными

6.1.4. Критерий эквивалентности блоков

Определение. Блок $\mathcal{B}$ называется приведенный, если не существует такой блок $\mathcal{B}'$,что $\mathcal{B} \Rightarrow_{1,2} \mathcal{B}'$

Приведенный блок не содержит ни бесполезных операторов, ни избыточных вычислений

Если дан блок $\mathcal{B}$, то можно найти эквивалентный ему приведенный блок повторно применяю T_1 и T_2. Поскольку каждое применение T_1 и T_2 сокращает длину блока, в конце концов мы должны прийти к приведенному блоку. Для приведенных блоков $\mathcal{B}_1 \equiv \mathcal{B}_2$, тогда и только тогда, когда $D(\mathcal{B}_1) = D(\mathcal{B}_2)$. Таким образом если дан блок $\mathcal{B}$, то можно найти единственный граф, соответствующий всем приведенным блокам, получаемым из $\mathcal{B}$, независимо от конкретной последовательности преобразований T_1 и T_2, в результате которой был получен приведенный блок.

Определение. Пусть $P=S_1 \ldots S_n$ – список операторов блока. Обозначим через $E(P)$ множество выражений вычисляемых в P. Формально $E(P) = \{v_t(A) \mid S_t$ – осуществляет присвоение переменной A, $1 \le t \le n\}$.

Выражение η вычисляется в P κ раз, если существует κ таких различных значений t, что $v_t(A) = n$ и S_t присваивает значение A.

Лемма. Если $\mathcal{B} = (P, I, V)$ – приведенный блок, то P не вычисляет никакого выражения более одного раза.

Лемма. Если $\mathcal{B}_1 = (P_1, I_1, U_1)$ и $\mathcal{B}_2 = (P_2, I_2, U_2)$- эквивалентные приведенные блоки, то $E(P_1) = E(P_2)$.

Теорема. Если $\mathcal{B}_1$ и $\mathcal{B}_2$ – два приведенных блока, то $\mathcal{B}_1 \equiv \mathcal{B}_2$, тогда и только тогда, когда $D(\mathcal{B}_1) = D(\mathcal{B}_2)$.

Следствие. Все приведенные блоки эквивалентные данному имеют один и тот же граф.

Собирая все приведенные выше определения, можно доказать, что для того, чтобы превратить блок в любой ему эквивалентный достаточно четырех введенных преобразований.

Теорема. $\mathcal{B}_1 = \mathcal{B}_2$, тогда и только тогда, когда $\mathcal{B}_1 \Leftrightarrow_{1,2,3,4} \mathcal{B}_2$.

6.1.5. Оптимизация блоков

Основная задача оптимизации преобразовать блок $\mathcal{B}$, в блок $\mathcal{B}'$, оптимальный относительно некоторой оценки блока (рис. 11.3).

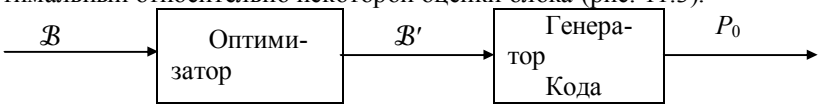


Рис. 11.3. Схема оптимизации.

Т.е. по данному блоку $\mathcal{B}$ мы хотим получить программу на объектном языке, оптимальную относительно некоторой оценки объектных программ, такой, например, как размер программы или время ее вы-

полнения. Оптимизатор применяет к блоку $\mathcal{B}$ последовательность преобразований, чтобы построить $\mathcal{B}'$, эквивалентный $\mathcal{B}$, из которого можно получить оптимальную программу. Таким образом, одна из задач заключается в оценки блоков.

Определение. Оценка блоков - это функция отображающая блоки в вещественные числа. Блок $\mathcal{B}$ называется оптимальным относительно оценки C, если $C(\mathcal{B}) \le C(\mathcal{B}')$, для всех $\mathcal{B}'$ эквивалентных $\mathcal{B}$. Оценка называется приемлемой, если $\mathcal{B}_1 \Rightarrow_{1,2} \mathcal{B}_2$ влечет $C(\mathcal{B}_2) \le C(\mathcal{B}_1)$ и любой блок имеет эквивалентный блок, оптимальный относительно C. Иными словами, оценка приемлема, если преобразования T_1 и T_2 применяемые в прямом направление, не увеличивают оценки блока.

Лемма. Если C - приемлемая оценка, то любой блок имеет эквивалентный приведенный блок, оптимальный относительно C.

Теорема. Пусть $\mathcal{B}$ - любой блок. Существует блок $\mathcal{B}'$ эквивалентный $\mathcal{B}$ и такой, что, если C - приемлемая оценка, то найдутся такие блоки, что

1) $\mathcal{B}' \Leftrightarrow_4 \mathcal{B}_1$,

2) $\mathcal{B}_1 \Leftrightarrow_3 \mathcal{B}_2$,

3) $\mathcal{B}_2$ оптимальный относительно C.

Таким образом, если мы хотим оптимизировать данный блок $\mathcal{B}$:

1) сначала можно исключить из $\mathcal{B}$ лишние и бесполезные вычисления и переименовать переменные с тем, чтобы получить приведенный открытый блок $\mathcal{B}'$,

2) затем в $\mathcal{B}'$ можно переупорядочить операторы с помощью перестановки и делать это до тех пор, пока не сформируется блок $\mathcal{B}_1$, в котором операторы расположены в наилучшем порядке,

3) наконец, в $\mathcal{B}_1$ можно переименовать переменные до тех пор, пока не будет найден оптимальный блок $\mathcal{B}_2$.

Пример. Рассмотрим машинный код, генерируемый для блоков. Вычислительная машина имеет один сумматор и следующий набор команд ассемблера:

1) LOAD M – содержимое ячейки памяти M помещается на сумматор.

2) STORE M – содержимое сумматора, запомнить в ячейке памяти M.

3) θ $M_2M_3\ldots M_r$. В этой команде θ - имя r-местной операции. Ее первый аргумент - сумматор, второй- ячейка памяти M_2 и т.д. Результат применения операции θ к своим аргументам размещается на сумматоре.

Генератор кода должен переводить оператор вида $A\leftarrow\theta B_1B_2\ldots B_r$ в последовательность машинных команд

LOAD B_1
θ $B_2, B_3\ldots B_r$
STORE A.

Однако если значение B_1 уже находится на сумматоре (т.е. перед этим было присвоение значения B_1), то первую команду LOAD генерировать не надо. Аналогично, если значение A не требуется негде, кроме первого аргумента следующего оператора, то команда STORE не нужна.

Оценить оператор $A\leftarrow\theta B_1\ldots B_r$ можно числами 1,2,3. Оценка равна 3, если B_1 нет на сумматоре и в дальнейшим есть ссылка на новое значение A, отличная от первого аргумента следующего оператора (т.е. A надо запомнить). Оценка равна 1, если B_1 уже на сумматоре и нет ссылок на A, отличной от первого аргумента следующего оператора. В остальных случаях оценка равна 2.

Рассмотрим блок $\mathcal{B}_1=(P, \{A, B, C\}, \{F, G\})$
$F=(A+B)*(A-B)$
$G=(A-B)*(A-C)*(B-C)$
Список операторов P_1 таков:

$T\leftarrow A+B$
$S\leftarrow A-B$
$F\leftarrow T*S$
$T\leftarrow A-B$
$S\leftarrow A-C$
$R\leftarrow B-C$
$T\leftarrow T*S$
$G\leftarrow T*R$.

Бесполезных операторов нет, но есть повторения - второй и четвертый операторы. Эту избыточность можно удалить. В результате получим приведенный операторный блок $\mathcal{B}_2=(P_2, \{A, B, C\}, \{X_3, X_7\})$

$X_1\leftarrow A+B$
$X_2\leftarrow A-B$
$X_3\leftarrow X_1-X_2$
$X_4\leftarrow A-C$

$X_5 \leftarrow B\text{-}C$
$X_6 \leftarrow X_2 * X_4$
$X_7 \leftarrow X_6 * X_5$

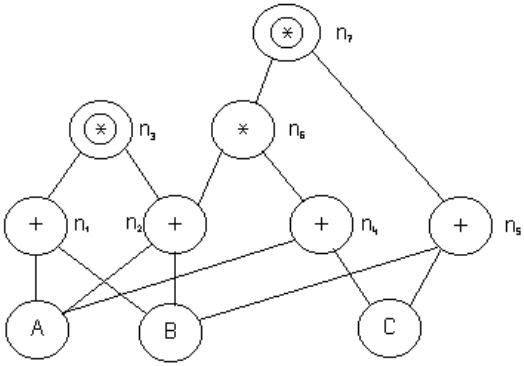


Рис. 11.4. Граф для $\mathcal{B}_2$.

Граф для $\mathcal{B}_2$ изображен на рис. 11.4. Существует много программ, в которые можно преобразовать $\mathcal{B}_2$ с помощью T_4.

Следующий алгоритм дает линейное упорядочивание вершин графов. В требуемом блоке операторы, соответствующие этим вершинам расположены в обратном порядке.

1) Строим список L. В начале он пуст.

2) Выбираем вершину n графа так, что $n \notin L$ и, если существует входящие в n дуги, они должны выходить из вершин уже принадлежащих к L. Если такой вершины нет, то остановится.

3) Если n_1- последняя вершина, добавляемая к L, самая левая дуга, выходящая из n_1, указывает на внутреннюю вершину n, не принадлежащую L, и все прямые предки n уже принадлежат L, то добавляем n к L и повторяем шаг 3), в противном случае переходим на шаг 2).

На приведенном графе можно начать с $L = n_3$. Согласно шагу 3), к L добавляем n_1. Затем выбираем и добавляем к L вершину n_7, а потом n_4 и n_5, так что L имеет вид n_3, n_1, n_7, n_6, n_2, n_4, n_5. Оператор, присваивающий значение X_i, порождает вершину n_i, и что список L соответствует операторам в обратном порядке, получаем блок $\mathcal{B}_3=(P_3, \{A, B, C\}, \{X_3, X_7\})$, где P_3

$X_5 \leftarrow B\text{-}C$
$X_4 \leftarrow A\text{-}C$
$X_2 \leftarrow A\text{-}B$

$X_6 \leftarrow X_2 * X_4$
$X_7 \leftarrow X_6 * X_5$
$X_1 \leftarrow A + B$
$X_3 \leftarrow X_1 * X_2$.

LOAD A	LOAD B
ADD B	SUBTR C
STORE X	STORE X_5
LOAD A	LOAD A
SUBTR B	SUBTR C
STORE X_2	STORE X_4
LOAD X_1	LOAD A
MULT X_2	SUBTR B
STORE X_3	STORE X_2
LOAD A	MULT X_4
SUBTR C	MULT X_5
STORE X_4	STORE X_7
LOAD B	LOAD A
SUBTR C	ADD B
STORE X_5	MULT X_2
LOAD X_2	STORE X_3
MULT X_4	
MULT X_5	
STORE X_7	
а – из $\mathcal{B}_2$	*б – из* $\mathcal{B}_3$

6.1.6. Алгебраические преобразования

Во многих языках программирования некоторые операции и операнды подчиняются определенным алгебраическим законам. Учитывая эти законы, можно проводить такие улучшения программы, какие невозможно сделать с использованием четырех рассмотренных выше типов преобразований.

Рассмотрим наиболее распространенные алгебраические преобразования.

1) Бинарная операция θ называется *коммутативной*, если $\alpha\theta\beta = \beta\theta\alpha$ для всех выражений α и β. Примером коммутативных операций случат сложение и умножение чисел.

2) Бинарная операция θ называется *ассоциативной*, если $\alpha\theta(\beta\theta\gamma) = (\alpha\theta\beta)\theta\gamma$ для всех α, β и γ. Например сложение коммутативно, так как $\alpha+(\beta+\gamma) = (\alpha+\beta)+\gamma$.

3) Бинарная операция θ_1 называется *дистрибутивной* относительно бинарной операции θ_2, если $\alpha\theta_1(\beta\theta_2\gamma) = (\alpha\theta_1\beta)\theta_2(\alpha\theta_1\gamma)$. Например, умножение дистрибутивно относительно сложения, так как $\alpha*(\beta+\gamma) = \alpha*\beta+\alpha*\gamma$.

4) Унарная операция θ называется *идемпотентной*, если $\theta\theta\alpha = \alpha$ для всех α. Например, логическое *не* и унарная операция «минус» идемпотентны.

5) Выражение ε называется *нейтральным* относительно бинарной операции θ, если $\varepsilon\theta\alpha = \alpha\theta\varepsilon = \alpha$ для всех α.

(а) Константа 0 нейтральна относительно сложения. Нейтрально и любое выражение, имеющее значение 0, Например, $\alpha - \alpha$, $\alpha * 0$, $(-\alpha) + \alpha$.

(б) Константа 1 нейтральна относительно умножения.

(в) Логическая константа *истина* нейтральна относительно конъюнкции (т.е. $\alpha \wedge$ *истина* $= \alpha$ для всех α).

(г) Логическая константа *ложь* нейтральна относительно дизъюнкции (т.е. $\alpha \vee$ *ложь* $= \alpha$ для всех α).

Если $\mathcal{A}$ – множество алгебраических законов, будем говорить, что *выражение* α *эквивалентно выражению* β *относительно* $\mathcal{A}$ (и писать $\alpha \equiv_{\mathcal{A}} \beta$), если α можно преобразовать в β, применяя только алгебраические законы $\mathcal{A}$.

Пример. Рассмотрим выражение

$A * (B * C) + (B * A) * D + A * E$.

С помощью ассоциативного закона умножения можно записать $A*(B*C)$ в виде $(A*B)*C$. С помощью коммутативного закона умножения можно записать $B*A$ в виде $A*B$. Применяя дистрибутивный закон, все выражение можно записать

$(A * B) * (C + D) + A * E$.

Наконец, применяя ассоциативный закон к первому слагаемому, затем дистрибутивный закон, можно записать выражение как

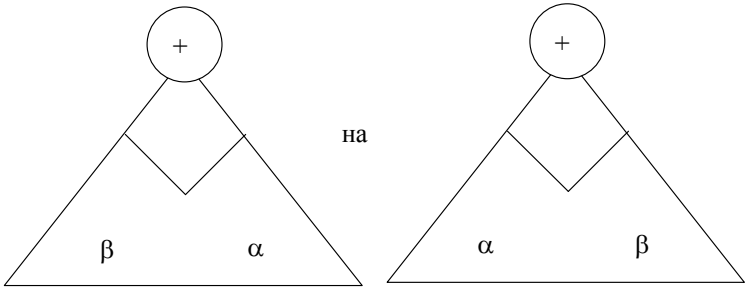
$(A * (B * (C + D) + E)$.

Таким образом, это выражение эквивалентно исходному относительно ассоциативного, коммутативного и дистрибутивного законов умножения и сложения. Одного оно вычисляется только двумя умножениями и двумя сложениями, в то время как для исходного требовалось пять умножений и два сложения.

Определение эквивалентности относительно множества алгебраических законов $\mathcal{A}$ можно распространить и на блоки. Будем говорить,

что блоки $\mathcal{B}_1$ и $\mathcal{B}_2$ *эквивалентны относительно* $\mathcal{A}$ (и писать $\mathcal{B}_1 \equiv_{\mathcal{A}} \mathcal{B}_2$), если существует такое выражение $\beta \in v(\mathcal{B}_2)$, что $\alpha \equiv_{\mathcal{A}} \beta$, и обратно.

Пример. Если сложение коммутативно, то преобразование блоков, соответствующее этому алгебраическому закону, позволяет заменять в блоке оператор вида $X \leftarrow A+B$ на оператор $X \leftarrow B+A$. Соответствующие преобразования графа позволяют заменить всюду в графе структуру



Если дан конечный набор алгебраических законов и соответствующие преобразования блоков, то для нахождения оптимального блока, эквивалентного данному, желательно было бы применять их вместе с топологическими преобразованиями. К сожалению, для конкретного набора алгебраических законов может не быть эффективного способа применения этих преобразований для нахождения оптимального блока.

Обычный подход к решению этого вопроса заключается в том, что алгебраические преобразования применяют в ограниченном виде, надеясь сделать больше «упрощений» выражений и выработать, возможно, большее число общих подвыражений. В типичной схеме $\beta\theta\alpha$ заменяется на $\alpha\theta\beta$, если θ - коммутативная бинарная операция, а α предшествует β при некотором лексикографическом упорядочении имен переменных. Если θ - ассоциативная и коммутативная бинарная операция, то $\alpha_1\theta\alpha_2\theta\ldots\theta\alpha_n$ можно преобразовав, располагая имена $\alpha_1,\ldots, \alpha_n$ в лексикографическом порядке и группирую их слева направо.

Пример. Рассмотрим блок $\mathcal{B} = (P, I, \{Y\})$, где $I = \{A, B, C, D, E, F\}$, а P – последовательность операторов

$$X_1 \leftarrow B\text{-}C$$
$$X_2 \leftarrow A*X_1$$

$$X_3 \leftarrow E*F$$
$$X_4 \leftarrow D*X_3$$
$$Y \leftarrow X_2*X_3$$

Блок $\mathcal{B}$ вычисляет выражение
$Y = (A * (B - C)) * (D * (E * F))$.

Граф для $\mathcal{B}$ приведен на рис. 11.5.

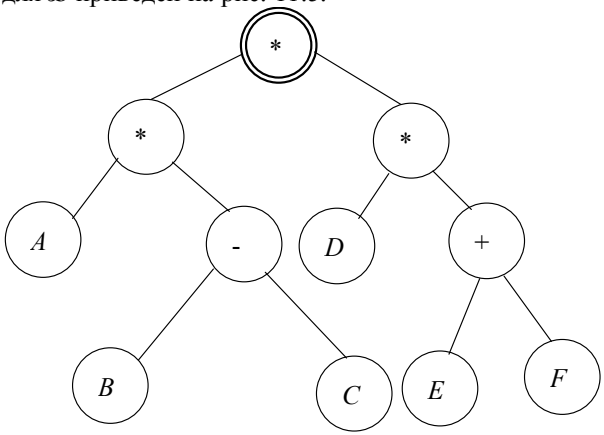


Рис. 11.5. Граф для $\mathcal{B}$.

Предположим, что для $\mathcal{B}$ генерируется программа на языке ассемблера и используются введенные нами ранее функции оценки. Применяя к $\mathcal{B}$ коммутативные и ассоциативные преобразования для умножения проведем последовательное преобразование программы.

Полагая, что умножения ассоциативно, можно заменить два оператора в $\mathcal{B}$

$$X_3 \leftarrow E*F$$
$$X_4 \leftarrow D*X_3$$

на три оператор

$$X_3 \leftarrow E*F$$
$$X_3' \leftarrow D*E$$
$$X_4 \leftarrow X_3'*F$$

Теперь оператор $X_3 \leftarrow E*F$ бесполезен, и его можно удалить преобразованием T_1. Затем с помощью ассоциативного преобразования можно заменить операторы

$$X_4 \leftarrow X_3'*F$$

$$Y \leftarrow X_2 * X_3$$

на

$$X_4 \leftarrow X_3' * F$$
$$X_4' \leftarrow X_2 * X_3'$$
$$Y \leftarrow X_4' * F$$

Оператор $X_4 \leftarrow X_3' * F$ теперь бесполезен, и его можно удалить. Таким образом имеем пять операторов

$$X_1 \leftarrow B\text{-}C$$
$$X_2 \leftarrow A * X_1$$
$$X_3' \leftarrow D * E$$
$$X_4' \leftarrow X_2 * X_3'$$
$$Y \leftarrow X_4' * F$$

Если применить ассоциативные преобразования еще раз к третьему и четвертому оператору, получим

$$X_1 \leftarrow B\text{-}C$$
$$X_2 \leftarrow A * X_1$$
$$X_3'' \leftarrow X_2 * D$$
$$X_4' \leftarrow X_3'' * E$$
$$Y \leftarrow X_4' * F$$

Наконец, если предположить, что умножение коммутативно, можно переставить операнды второго оператора и получить блок $\mathcal{B}'$:

$$X_1 \leftarrow B\text{-}C$$
$$X_2 \leftarrow X_1 * A$$
$$X_3'' \leftarrow X_2 * D$$
$$X_4' \leftarrow X_3'' * E$$
$$Y \leftarrow X_4' * F$$

Граф для $\mathcal{B}'$ изображен на рис. 11.6. Блок $\mathcal{B}'$ имеет оценку 7, самую нижнюю возможную оценку для исходной программы.

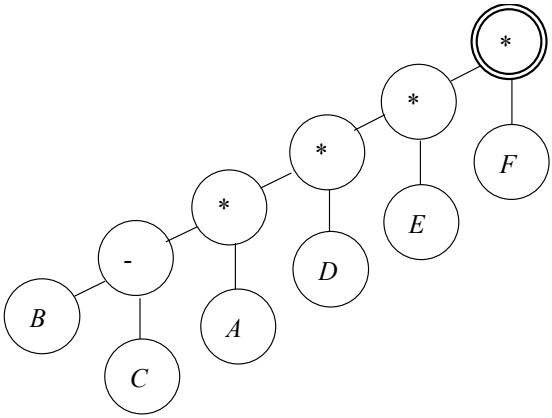


Рис.11.6. Граф $\mathcal{B}'$.

6.2. Арифметические выражения

Входом для любого генератора кода служит блок, состоящий из последовательности операторов присвоения. Выходом – эквивалентная программа на языке ассемблера.

Желательно, чтобы результирующая программа на языке ассемблера была бы «хорошей» относительно некоторой оценки, такой, например, как число команд ассемблера или число обращений к памяти.

В данном разделе мы будем рассматривать эффективный алгоритм генерации кода для ограниченного класса блоков, а именно блоков, представляющих собой одно выражение без одинаковых операндов. Предположение об отсутствии одинаковых операндов, естественно, не реально, но часто бывает весьма хорошим первым приближением.

Блок, представляющий одно выражение, имеет только одну выходную переменную. Ограничение, состоящее в том, что выражение не имеет одинаковых операндов, эквивалентно требованию, чтобы граф для этого выражения был деревом.

Для удобства будем предполагать, что все операции бинарные. Код на языке ассемблера будем генерировать для машины с N сумматорами, где $N \geq 1$. Оценкой будет длина программы (т.е. число команд).

6.2.1. Модель машины

Рассмотрим машину с $N \geq 1$ универсальными сумматорами и командами четырех типов.

Определение. Командой языка ассемблера называется цепочка символов одного из четырех типов:

LOAD M, A
STORE A, M
OP θ A, M, B
OP θ A, B, C.

В этих командах M – ячейка памяти, A, B, C – имена сумматоров (возможно одинаковые). OP θ - это код бинарной операции θ. Предполагается, что каждой операции θ соответствует машинная команда 3) или 4). Эти команды выполняют следующие действия.

1) LOAD M, A помещает содержимое ячейки памяти M на сумматор A.

2) STORE A, M помещает содержимое сумматора A в ячейку памяти M.

3) OP θ A, M, B применяет бинарную операцию θ к содержимому сумматора A и ячейки памяти M, а результат помещает на сумматор B.

4) OP θ A, B, C применяет бинарную операцию θ к содержимому сумматоров A и B, а результат помещает на сумматор C.

Программой на языке ассемблера будем называть последовательность команд языка ассемблера.

Если $P = I_1; I_2; \ldots; I_n$ – программа, то можно определить *значение* $v_t(R)$ *регистра* R *последней команды* t (под регистром понимается сумматор или ячейка памяти):

1) $v_0(R)$ равно R если R – это ячейка памяти, и не определено, если R – сумматор,

2) если I_t – это LOAD M, A, то $v_t(A) = v_{t-1}(M)$,

3) если I_t – это STORE A, M, то $v_t(M) = v_{t-1}(A)$,

4) если I_t – это OP θ A, B, C, то $v_t(C) = \theta\, v_{t-1}(A)\, v_{t-1}(R)$,

5) если $v_t(R)$ не определено по 2) – 4), а $v_{t-1}(R)$ определено, то $v_t(R) = v_{t-1}(R)$; в противном случае $v_t(R)$ не определено.

Команды LOAD и STORE передают значения с одного регистра на другой, оставляя их в исходном регистре. Операции перемещают вычисленное значение на сумматор, определенный третьим аргументом, не меняя остальных регистров. Будем говорить, что программа P *вычисляет* выражение α, помещая результат на сумматор A, если после последнего оператора из P сумматор A имеет значение α.

Пример. Рассмотрим программу на языке ассемблера с двумя сумматорами A и B, значения которых после каждой команды приведены в табл. 11.3.

Таблица 11.3.

	$v(A)$	$v(B)$
LOAD X, A	X	не определено
ADD A, Y, B	$X + Y$	не определено
LOAD Z, B	$X + Y$	Z
MULT B, A, A	$Z * (X + Y)$	Z

Значение сумматора A в конце программы соответствует выражению $Z*(X + Y)$. Таким образом, эта программа вычисляет $Z * (X + Y)$, помещая результат на сумматор A.

Формально определим *синтаксическое дерево* как помеченное двоичное дерево T, имеющих одну или более таких вершин, что

1) каждая внутренняя вершина помечена бинарной операцией $\theta \in \Theta$,

2) все листья помечены различными именами переменных $X \in \Sigma$.

Будем предполагать, что множества Θ и Σ не пересекаются. Вершинам дерева, начиная снизу, можно следующим образом приписать *значения*:

1) если n – лист, помеченный X, то n имеет значение X,

2) если n – внутренняя вершина, помеченная θ, и ее прямыми потомками являются n_1 и n_2 со значениями v_1 и v_2, то n имеет значение $\theta\ v_1\ v_2$.

Значением дерева будем считать значение его корня.

Можно естественным образом, оператор за оператором преобразовывать блок на промежуточном языке в программу на языке ассемблера. Важной составляющей преобразования является алгоритм разметки синтаксического дерева.

6.2.2. Разметка дерева

Алгоритм разметки синтаксического дерева.

Вход. Синтаксическое дерево T.

Выход. Помеченное синтаксическое дерево.

Метод. Вершинам дерева T рекурсивно, начиная снизу, назначаем целочисленные метки:

1) Если вершина есть лист, являющимся левым прямым потомком своего прямого предка, или корень, помечаем вершину 1; если вершина есть лист, являющимся правым потомком, помечаем ее 0.

2) Если вершина n имеет прямых потомков n_1 и n_2, помеченных l_1, l_2. Если $l_1 = l_2$. берем в качестве метки число $l_1 + 1$.

Пример. Арифметическое выражение

$A*(B-C)/D*(E-F)$

изображенное в виде дерева на рис. 11.7, на котором представлены целочисленные метки.

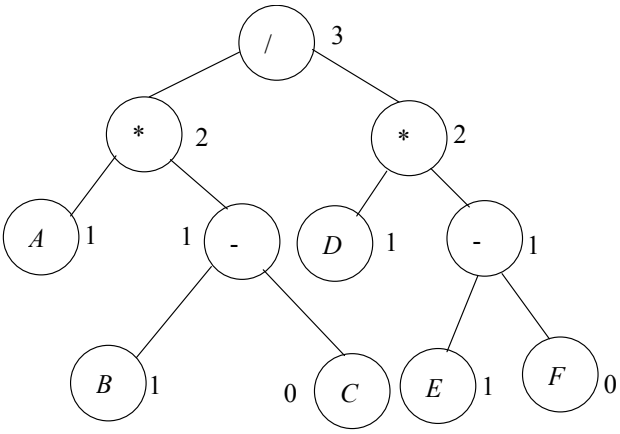


Рис. 11.7. Помеченное синтаксическое дерево.

Рассмотрим алгоритм, преобразующий помеченное синтаксическое дерево в программу на языке ассемблера для машины с N сумматорами. Для любого N можно построить, что получаемая программа оптимальна относительно различных оценок.

Алгоритм построения кода на языке ассемблера для выражений.

Вход. Помеченное синтаксическое дерево T и N сумматоров A_1, A_2, …, A_N, где $N \geq 1$.

Выход. Программа P на языке ассемблера, после последней команды которой значением $v(A_1)$ становится $v(T)$; т.е. P вычисляет выражение, представленное деревом T, и помещает результат на сумматор A_1.

Метод. Предполагается, что дерево T помечено в соответствии с алгоритмом разметки синтаксического дерева. Затем рекурсивно выполняется процедура **code**(n, i). Входом для **code** служит вершина n дерева T и целое число i от 1 до N. Число i означает, что в данный момент для вычисления выражения доступны сумматоры A_i, A_{i+1}, …, A_N. Выходом **code**(n, i) служит последнее значение $v(n)$ и помещает результат на сумматор A_i.

Сначала выполняется **code**(n_0, 1), где n_0 – корень дерева T. Последовательность команд, генерированных этим вызовом процедуры **code,** и будет нужной программой на языке ассемблера.

Процедура **code**(n, i). Предполагается, что n - вершина дерева T, а i - целое число между 1 и N.

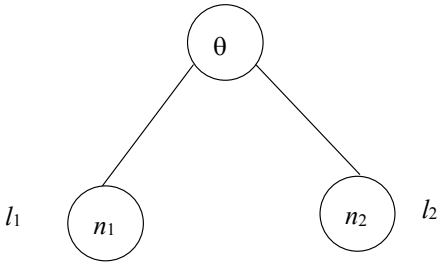
1) Если n – лист, выполняем шаг 2). В противном случае выполняем шаг 3).

2) Если вызвана процедура **code**(n, i), а n – лист, то n всегда будет левым прямым потомком (или корнем, если n – единственная вершина дерева). Если с листом n связано имя переменной X, то

code(n, i) = ‘LOAD X, A_i’

(это означает, что выходом процедуры **code**(n, i) служит команда LOAD X, A_i).

3) В это место мы приходим, только если n – внутренняя вершина. Пусть с вершиной n операция θ и ее прямыми потомками являются n_1 и n_2 с метками l_1 и l_2.



Следующий шаг определяется значением меток l_1 и l_2:

(а) если $l_2 = 0$ (n_2 – правый лист), выполняем шаг 4),

(б) если $1 \le l_1 < l_2$ и $l_1 < N$, выполняем шаг 5),

(в) если $1 \le l_2 \le l_1$ и $l_2 < N$, выполняем шаг 6),

(г) если $N \le l_1$ и $N < l_2$, выполняем шаг 7).

4) **code**(n, i) = **code**(n_1, i)

‘OP θ A_i, X, A_i’

Здесь X – переменная, связанная с листом n_2, а OP θ - код операции θ. Выходом для **code**(n, i) служит выход для **code**(n_1, i), сопровождаемый командой OP θ A_i, X, A_i.

5) **code**(n, i) = **code**(n_2, i)

code(n_1, i+1)

‘OP θ A_{i+1}, A_i, A_i’

6) **code**(n, i) = **code**(n_1, i)

code(n_2, i+1)

‘OP θ A_i, A_{i+1}, A_i’

7) **code**(n, i) = **code**(n_2, i)
$T \leftarrow$ **newtemp**
'STORE A_i, T'
code(n_1, i)
'OP θ A_i, T, A_i'

Здесь **newtemp** – функция, которая при обращении к ней вырабатывает новую временную ячейку памяти для запоминания промежуточных результатов.

Пример. Применим алгоритм при N =2 к синтаксическому дереву на рис. 11.6. Последовательность вызовов **code**(n, i) приведена в табл. 11.4.

Таблица 11.4.

Вызов	Шаг
code(/, 1)	(3г)
code($*_R$, 1)	(3в)
code(D, 1)	(2)
code($-_R$, 2)	(3а)
code(E, 2)	(2)
code($*_L$, 1)	(3в)
code(A, 1)	(2)
code($-_L$, 2)	(3а)
code(B, 2)	(2)

Здесь $*_L$ – ссылка на левый потомок вершины /, $*_R$ – на правый потомок вершины $*_L$, а $-_R$ – на правый потомок вершины $*_R$.

Процедура **code**(/, 1) генерирует программу

```
LOAD   D, A1
LOAD   E, A2
SUBTR  A2, F, A2
MULT   A1, A2, A1
STORE  A1, TEMP1
LOAD   A1, A1
LOAD   B, A2
SUBTR  A2, C, A2
MULT   A1, A2, A1
DIV    A1, TEMP1, A1
```

Рассмотренные нами алгоритмы позволяют сформулировать две базовые теоремы для построения оптимального кода.

Теорема 1. Программа, выработанная процедурой code(n, i) правильно вычисляет значение вершины n, помещая его на i-й сумматор.

Теорема 2. Пусть *T* - синтаксическое дерево, *l* – метка его корня, *N* - число доступных сумматоров. Программа, вычисляющая T без использования команд STORE, существует тогда и только тогда, когда $L \leq N$.
Выясним теперь, сколько команд LOAD и STORE требуется для вычисления синтаксического дерева, если использовать *N* сумматоров, когда корень помечен числом превышающим *N*.

6.2.3. Программы с командами STORE

Определение. Пусть *T* - синтаксическое дерево, а *N* - число доступных сумматоров. Вершина из *T* называется старшей, если каждый прямой ее потомок помечен числом не меньше чем *N*. Вершина называется младшей, если она является листом и левым потомком своего прямого предка (т.е. лист с меткой 1).

На рис. 11.7. при *N*=2 – одна старшая вершина – корень и четыре младших – листья со значениями *A, B, D* и *E*.

Лемма 1. Пусть *T* - синтаксическое дерево. Программа *P*, вычисляющая *T* и использующая *m* команд LOAD, существует тогда и только тогда, когда *T* имеет не более *m* младших вершин.

Лемма 2. Пусть *T* - синтаксическое дерево. Программа *P*, вычисляющая *T* и использующая *M* команд STORE, существует тогда и только тогда, когда *T* имеет не более *M* старших вершин.

Теорема. Алгоритм построения кода на языке ассемблера всегда вырабатывает кратчайшую программу для вычисления данного выражения.

Доказательство. В соответствии с введенными леммами 1 и 2 алгоритм вырабатывает программу с минимальным числом команд LOAD и STORE. Поскольку ясно, что минимальное количество команд операций равно числу внутренних вершин дерева, а алгоритм дает по одной такой команде для каждой внутренней вершине дерева, то утверждение теоремы очевидно.

Для нашего примера имеется одна старшая и четыре младших вершины (*N*=2). Арифметическое выражение имеет пять внутренних вершин. Таким образом, для вычисления требуется не менее 10 операторов.

6.2.4. Влияние некоторых алгебраических законов

Определим *оценку синтаксического дерева* как сумму

1) числа внутренних вершин,
2) числа старших вершин,

3) числа младших вершин.

Результаты, изложенные выше, показывают, что эта оценка служит разумной мерой «сложности» синтаксического дерева, поскольку число команд необходимых для вычисления синтаксического дерева, равно его оценке.

Часто, к некоторым операциям можно применить алгебраические законы и с их помощью уменьшить оценку данного синтаксического дерева. Из рассмотренного нами раннее известно, что каждый алгебраический закон индуцирует соответствующее преобразование алгебраического дерева. Например, если *n*-внутренняя вершина синтаксического дерева, связанная с коммутативной операцией, то коммутативное преобразование меняет порядок прямых потомков вершины *n*.

Аналогично, если θ – ассоциативная операция (т.е. $\alpha\theta(\beta\theta\gamma) = (\alpha\theta\beta)\theta\gamma$), то, применяя соответствующие преобразования деревьев, можно перевести в друг друга два дерева вида, изображенные на рис. 11.8.

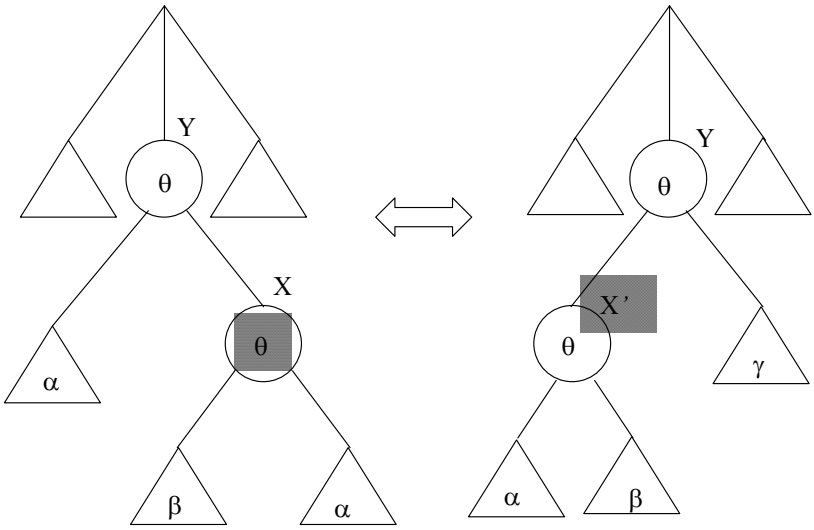


Рис. 11.8. Ассоциативное преобразование синтаксических деревьев.

Ассоциативное преобразование в этом случае имеет вид

$$\begin{matrix} X \leftarrow B\theta C \\ Y \leftarrow A\theta Y \end{matrix} \quad \Leftrightarrow \quad \begin{matrix} X' \leftarrow A\theta B \\ Y \leftarrow X'\theta C \end{matrix}$$

После проведения преобразования слева направо оператор $X \leftarrow B\theta C$ сохранился. Однако, после преобразования этот оператор становится бесполезным, так что его можно удалить, не меняя значение блока.

Определение. Если дано множество $\mathcal{A}$ алгебраических законов, то два синтаксических дерева T_1 и T_2 будут называться эквивалентными относительно A ($T_1 \equiv_{\mathcal{A}} T_2$), если существует последовательность преобразований, выводимая из этих законов, переводящих T_1 в T_2. Через $[T]_{\mathcal{A}}$ будем обозначать класс эквивалентности деревьев $\{T' \mid T \equiv_{\mathcal{A}} T'\}$.

Таким образом, если дано синтаксическое дерево T и известно, что выполняются алгебраические законы из некоторого множества законов $\mathcal{A}$, то для нахождения оптимальной программы для T надо искать в $[T]_{\mathcal{A}}$ дерево выражений с минимальной оценкой. Если дерево с минимальной оценкой найдено, то для нахождения оптимальной программы можно применить алгоритм построения оптимального кода на языке ассемблера.

Если каждый закон сохраняет число операций, как в случае коммутативного и ассоциативного закона, достаточно минимизировать сумму чисел старших и младших вершин.

Рассмотрим такой алгоритм. Сначала для случая, когда коммутативные операции также и ассоциативны.

По данному синтаксическому дереву и множеству $\mathcal{A}$ алгебраических законов приведенный ниже алгоритм будем строить синтаксическое дерево $T \in [T]_{\mathcal{A}}$ с минимальной оценкой при условии , что $\mathcal{A}$ содержит только ассоциативные законы, применяемые к определенным операциям. Затем к T' можно будет применить алгоритм построения кода на языке ассемблера для выражений. и найти оптимальную программу для исходного дерева.

Алгоритм построение дерева с минимальной оценкой, некоторые операции которого коммутативны.

Вход. Синтаксическое дерево T (с тремя и более вершинами) и множество коммутативных законов.

Выход. Синтаксическое дерево $[T]_{\mathcal{A}}$ с минимальной оценкой.

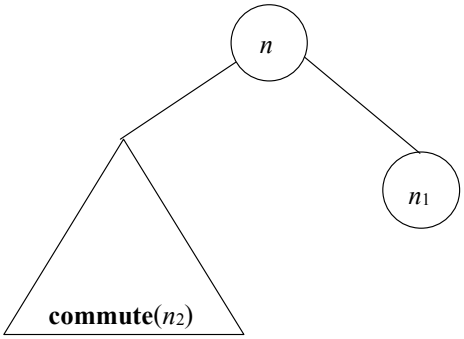
Метод. Ядро алгоритма составляет рекурсивная процедура **commute**(n)**,** аргументом которой служит вершина n синтаксического дерева, а результатом – модифицированное поддерево с вершиной n в качестве корня. В начале вызывается **commute**(n_0), где n_0-корень данного дерева T.

Процедура **commute**(n)**.**

1) Если *n*-лист, полагаем **commute**(*n*)=*n*

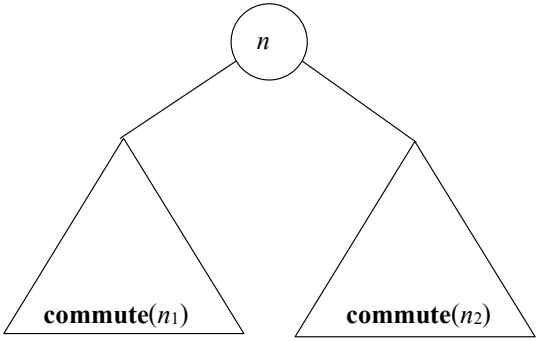
2) Если *n*-внутренняя вершина, рассмотрим два случая.

(а)Пусть вершина *n* имеет два прямых потомка n_1 и n_2 (в указанном порядке) и операция связанная *n* коммутативна. Если n_1- лист, а n_2 нет, то выход **commute**(*n*) - дерево типа *а*).



а

Во всех остальных случаях выход **commute**(*n*) - дерево типа *б*).



б

Пример. Рассмотрим рис. 11.7. и предположим, что коммутативно только умножение. Результат применения алгоритма к этому дереву показан на рис. 11.9.

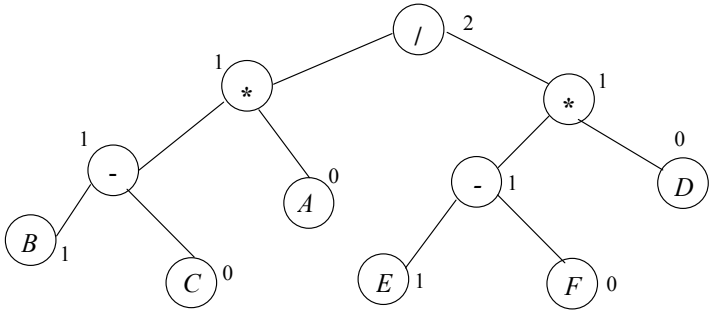


Рис. 11.9. Преобразованное арифметическое выражение.

Отметим, что корень дерева помечен 2 и здесь лишь две младшие вершины. Таким образом, если у нас два сумматора, то на вычисление этого дерева требуется только 7 команд (а для исходного 10).

Теорема. Если допустимо применение только одного коммутативного для некоторых операций закона, то алгоритм построение дерева с минимальной оценкой, некоторые операции которого коммутативны, вырабатывает такое синтаксическое дерево из класса эквивалентности для данного дерева, которое имеет минимальную оценку.

Доказательство. Легко видеть, что коммутативный закон не изменяет числа внутренних вершин. Простая индукция по высоте вершины показывает, что алгоритм минимизирует число младших вершин и метки, которые должны быть у вершин после применения алгоритма разметки. Следовательно, минимизирует и число старших вершин.

Ситуация усложняется, когда некоторые операции одновременно коммутативны и ассоциативны. В этом случае уменьшить число старших вершин можно за счет существенного преобразования дерева.

Определение. Пусть T - синтаксическое дерево. Множество S из двух или более его вершин называется кистью, если

1) все вершины в S внутренние и представляют одну и ту же ассоциативную и коммутативную операцию,

2) вершины из S вместе с соединяющими дугами образуют дерево,

3) ни одно из собственных подмножеств множества S не обладает свойствами 1) и 2).

Корнем кисти называется корень дерева, о котором идет речь в пункте 2).

Прямыми потомками кисти S называются те вершины из T, которые не принадлежат S, но являются прямыми потомками вершин из S.

Пример. Рассмотрим синтаксическое дерево на рис. 11.10 в котором, слоение и умножение коммутативны, а никакие другие операции не применимы.

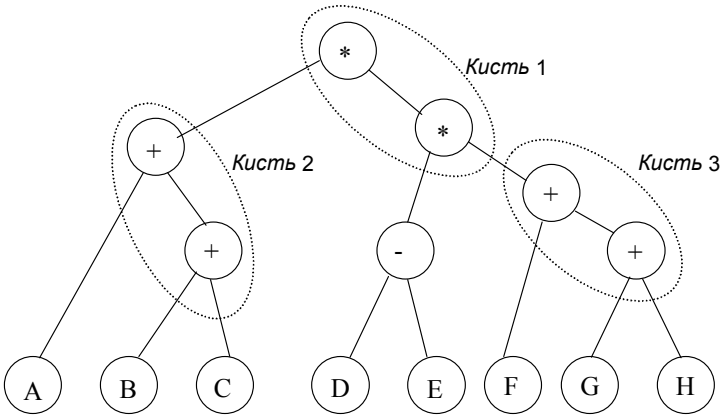


Рис. 11.10. Синтаксическое дерево.

Кисти синтаксического дерева T определяются однозначно и не пересекаются. Для нахождения $[T]_{\mathcal{A}}$ дерева минимальной оценкой, когда $\mathcal{A}$ содержит законы, отражающие тот факт, что одни операции коммутативны и ассоциативны, а другие только коммутативны, вводится понятие ассоциативного дерева, в котором кисти представляются одной вершиной.

Определение. Пусть T – синтаксическое дерево. Ассоциативным деревом T' для T назовем дерево, полученное заменой каждой кисти S в T на единственную вершину n с той же ассоциативной и коммутативной операцией, что и у вершин кисти S. Прямые потомки кисти в T становятся прямыми потомками вершины n в T'.

Пример. Рассмотрим синтаксическое дерево на рис. 11.11. Предполагая, что сложение умножения и сложения коммутативны и ассоциативны, получим кисти.

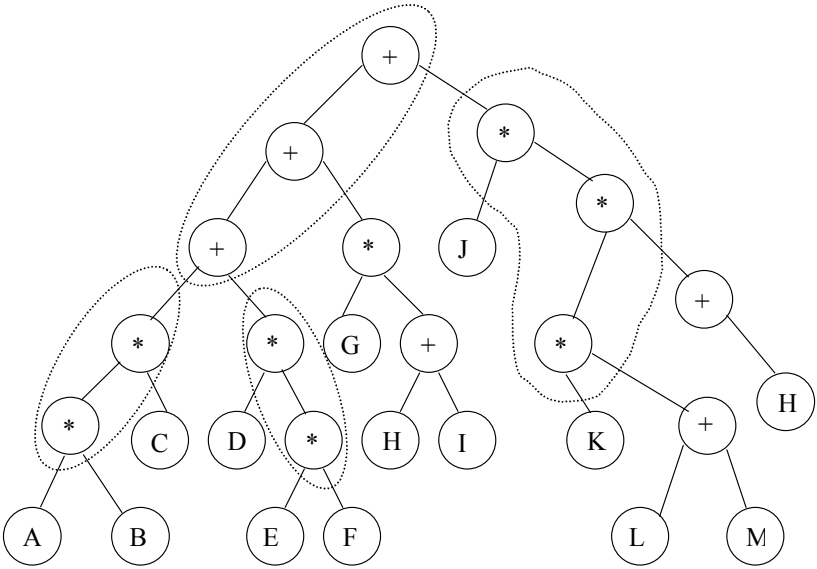


Рис. 11.11. Синтаксическое дерево с кистями.

Ассоциативное дерево для дерева *T* изображено на рис.11.12.

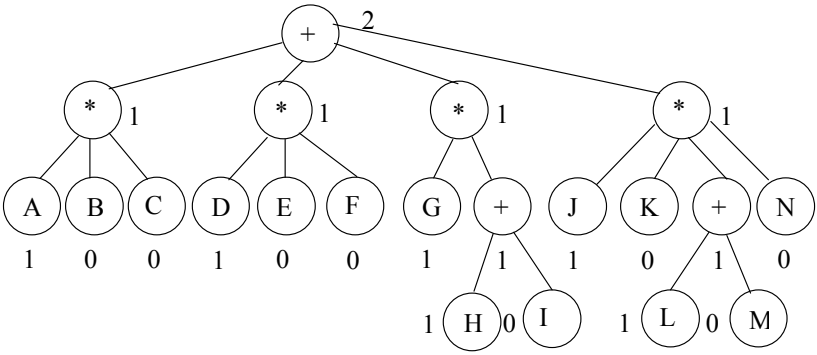


Рис. 11.12. Помеченное ассоциативное дерево.

Отметим, что ассоциативное дерево не обязательно должно быть двоичным.

Вершины ассоциативного дерева можно пометить целыми, начиная снизу следующим образом:

1) Лист, являющийся самым левым прямым потомком своего предка, помечен 1. Все остальные листья помечаются 0.

2) Пусть n - внутренняя вершина, прямые потомки которой n_1, n_2,.., n_m, помечаем l_1, l_2,.., l_m при m>=2.

a) Если одно из чисел l_1, l_2,.., l_m превосходит остальные, берем его в качестве метки вершины n.

b) Если вершина n имеет коммутативную операцию и n_i внутренняя вершина с l_i=1, а остальные вершины n_1, n_2,.., n_i -1, n_i+1,.., n_m-листья, то в качестве метки вершины n берем 1.

c) Если условие b) не выполняется , l_i=l_j для некоторых $i \neq j$ и l_i меньше остальных l_k, в качестве метки вершины n берем l_i+1.

Алгоритм построения синтаксического дерева с минимальной оценкой в предположении, что одни операции коммутативны, другие коммутативны и ассоциативны и больше никакие алгебраические законы не учитываются.

Вход. Синтаксическое дерево T и множество A коммутативных и коммутативно-ассоциативных законов.

Выход. Синтаксическое дерево $[T]_A$ с минимальной оценкой.

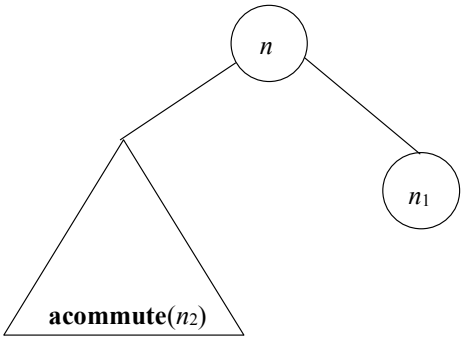
Метод. Строим сначала помеченное ассоциативное дерево T' для T. Затем вычисляем **acommute**(n_0), где **acommute** - процедура, определяемая ниже, а n_0 - корень дерева T'. Выходом **acommute**(n_0) служит синтаксическое дерево $[T]_A$ с минимальной оценкой.

Процедура **acommute**(n).

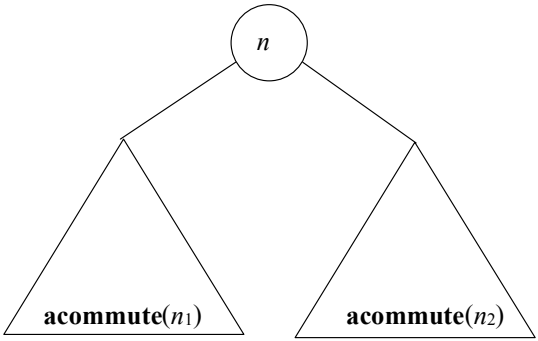
Аргументом n-служит вершина помеченного ассоциативного дерева. Если n - лист, полагаем **acommute**(n)=n. Если n - внутренняя вершина, то рассмотрим три случая:

1) Пусть вершина имеет два прямых потомка n_1 и n_2 и операция связанная с n коммутативна (и, возможно ассоциативна)

а) Если n_1 - лист, а n_2 нет, то выход **acommute**(n)-дерево



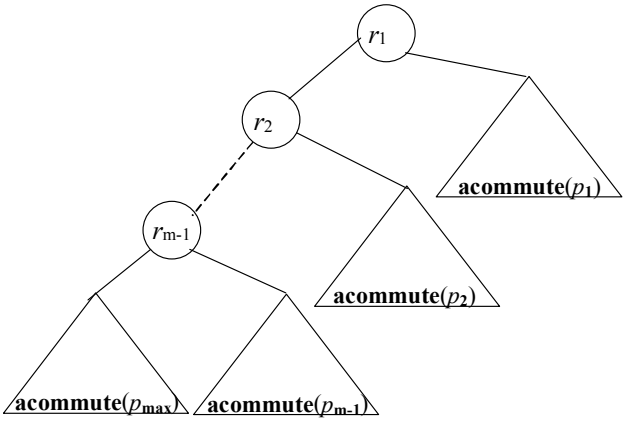
б) в противном случае **acommute**(n) дерево



2) Предположим, что операция θ, связанная с n коммутативна и ассоциативна и вершина n имеет прямых потомков n_1, n_2,.., n_m, m>=3 (в порядке слева направо)

Пусть n_{max} – вершина из n_1, .., n_m с наибольшей меткой. Если одной и той же наибольшей меткой помечается две или более вершин, то вершину n_{max} выбираем так, чтобы она была внутренней. Обозначим через p_1, p_2,.., p_{m-1} вершины в $\{n_1, .., n_m\}$-$\{n_{max}\}$ в любом порядке.

Тогда выходом **acommute**(n) служит двоичное дерево, где r_i ($1 \le i \le m$-1) - новые вершины, связанные с коммутативной и ассоциативной операцией θ, соответствующей вершине n.



3) Если операция, связанная с n, некоммутативная и неассоциативна, то выходом **acommute**(n) служит дерево рис.

Применим алгоритм к помеченному ассоциативному дереву рис 11.12.

Применяя **accommute**, а точнее случай 2 к корню, берем в качестве n_{max} первого слева прямого потомка. В результате получим дерево рис. 11.13.

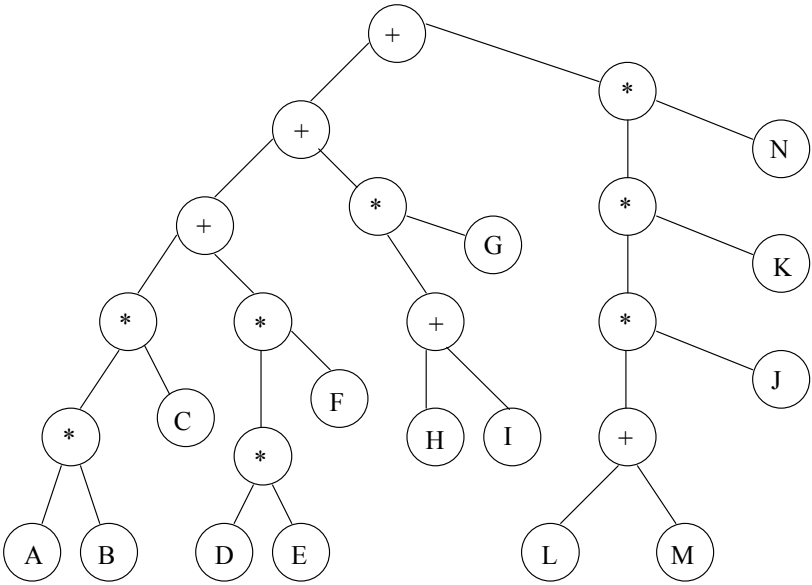


Рис. 11.13. Выход алгоритма.

Для доказательства оптимальности алгоритма введем несколько определений.

Лемма. Пусть T - помеченное дерево, а S-кисть в T. Предположим, что метки r прямых потомков вершин и S больше или равны N, где N - число сумматоров. Тогда по крайней мере r-1 вершин из S являются старшими.

Доказательство. Пусть n-корень кисти S и пусть n имеет потомков n_1 и n_2.

Случай 1. n_1 и n_2 не принадлежат S. Очевидно, что в этом случае утверждение верно (T_1 и T_2 поддеревья с корнями n_1 и n_2).

Случай 2. Пусть n_1 принадлежит S, а n_2 нет. Поскольку число вершин в дереве T_1 меньше чем в T, к нему применимо предположение индукции. Таким образом T_1 множество S-$\{n\}$ содержит по крайней мере r-2 старших вершин, если $l_2 \geq N$ и по крайней мере r-1 старших вершин, если $l_2 < N$. В последнем случае это тривиально. Рассмотрим случай $r>1$ и $l \geq N$. Тогда S-$\{n\}$ имеет по крайней мере одного прямого потомка, метка которого больше или равна N, так, что $l_1 \geq N$. Таким образом, n - старшая вершина и S содержит не менее r-1 старших вершин.

Случай 3. n_2 принадлежит S, а n_1 нет. Этот случай аналогичный 2.

Случай 4. n_1 и n_2 не принадлежит S. Пусть r_1 прямых потомков вершин из S с метками, не меньшими N, являются потомками вершины T, а r_2 из них являются потомками n_2. Тогда r_1-r_2=r. По предположению индукции части кисти S в T_1 и T_2 имеют соответственно не менее r_1-1 и не менее r_2-1 старших вершин. Если r_1 и r_2 не равны нулю, то $l_1 \geq N$, и $l_2 \geq N$, так что n-старшая вершина. Таким образом, S имеет по крайней мере (r_1-1)+(r_2-1)+1=r-1 старших вершин. Если r_1=0, то r_2=r, так что часть кисти S в T_2 имеет не менее r-1старших вершин. Случай r_2=0 аналогичен.

Теорема. Рассмотренный выше алгоритм вырабатывает дерево с минимальной оценкой.

Доказательство. Прямая индукция по числу вершин в ассоциативном дереве A показывает, что в результате применения процедуры **accommute** к его корню будет построено синтаксическое дерево T, корень которого после разметки помечен так же как и корень дерева A. Никакое дерево из $[T]_A$ не имеет корня с меткой , меньше чем в A, старших и младших вершин.

Предположим, что это не так. Тогда пусть T - наименьшее дерево, для которого одно из этих условий нарушается. Пусть θ – операция в корне дерева.

Случай 1. Операция θ не ассоциативна и не коммутативна. Любое ассоциативное и коммутативное преобразование дерева T должно совершиться целиком внутри поддерева, корнем которого служит один из прямых потомков корня T. Таким образом, касается ли нарушение метки, начала старших или младших вершин, оно должно проявиться в одном из этих поддеревьев, что противоречит минимальности дерева T.

Случай 2. Операция θ коммутативна, но не ассоциативна. Этот случай аналогичен случаю 1, но теперь коммутативное преобразование можно применить к корню. На шаге 1) процедуры **accomute** уже учитывалась возможность применения этого преобразование, так что любые нарушения дерева T вновь приведут к нарушению в одном из его поддеревьев.

Случай 3. Операция θ коммутативна и ассоциативна. Пусть S - кисть, содержащая корень. Можно считать, что ни в одном из деревьев, корнями которого служат потомки вершины из S, ни нарушается, ни одно из указанного выше условий. Любое ассоциативное или коммутативное преобразование совершается целиком внутри одного из этих поддеревьев или целиком внутри S. Ясно, что число младших

вершин, возникающих из S, минимально и значит метка корня минимальна.

6.3. Программы с циклами

При рассмотрении программ, содержащих циклы, трудно реализовать автоматическую минимизацию. Основные трудности здесь связаны с проблемами неразрешимости. Для двух произвольных программ не существует алгоритма, выясняющий эквивалентны ли в каком либо смысле. Как следствие этого не существует алгоритма, который находил бы по данной программе эквивалентную ей оптимальную оценку.

Это понятно, хотя бы по тому, что существует, вообще говоря, сколь угодно много способов вычисления одной и той же функции.

Тем не менее, во многих ситуациях можно указать набор преобразований, применимых к исходной программе для уменьшения размера и/или увеличения скорости результирующей объектной программы. Мы будем использовать преобразования, занимаясь улучшением кода, а не его оптимальности.

Главная цель – уменьшение времени выполнения объектной программы.

6.3.1. Модель программы

Мы будем использовать для программы представление, промежуточное между исходной программой и языком ассемблера. Программа состоит из последовательности операторов, за которым следует двоеточие.

Существует пять основных типов операций: присвоение, переход, условный переход, ввод-вывод и останов.

1) *Оператор присвоения* – это условие вида $A \leftarrow \Theta\, B_1\, B_2\, .\, .\, B_r$, где A - переменная, B_1, B_2,. . B_r – переменные либо константs, θ - r местная операция.

2) *Оператор перехода*

goto <метка>

где <метка>- цепочка букв. Будем предполагать, что одна и та же метка может быть только у одного оператора программы.

3) *Условный оператор*

if A <условие> B **goto** <метка>

где A и B - переменные или константы, а <отношение >- бинарное отношение, такое как <,<=,=,≠.

4) *Оператор ввода-вывода* – это либо оператор чтения вида

read *A*

где *A* переменная, либо оператор записи

write *B*

где *B* - переменная, либо константа.

5) *Оператор останова* – это команда **halt**.

Пример:

if *A r B* **goto** *L*

означает, что между текущими значениями *A* и *B* выполняется отношение *r*, то управление передается оператору, помеченному *L*. В противном случае управление следующему оператору.

Оператор определения – это оператор вида **read** *A* или $A \leftarrow \theta\, B_1\, B_2\, .\, .\, B_r$.

Оба этих оператора *определяют* переменную *A*.

Сделаем несколько предположений.

Переменная – это либо простая переменная, либо индексирование простой переменной, либо константой, например $A(1)$, $A(2)$, …, $A(i)$, либо $A(j)$.

Далее будем считать, что все переменные, на которые в программе есть ссылки, либо должны быть входными переменными, либо должны быть раннее определяться с помощью оператора присвоения.

Наконец будем предполагать, что каждая программа содержит хотя бы один оператор останова.

Выполнение программы начинается с первого оператора и продолжается пока не встретится оператор останова.

Будем считать , что входные переменные программы – это те, которые связаны с оператором чтения, а выходные – с оператором записи.

Присвоение значения каждой переменной при каждом чтении называется входным присвоением. Значение программы – это последовательность значений записанных входными переменными в процессе выполнения программы. Две программы будут считаться эквивалентными, если для каждого входного присвоения они имеют одинаковые значения.

Это определение эквивалентности обобщает эквивалентность блоков. Предположим, что два блока $\mathcal{B}_1=(P_1, I_1, U_1)$ и $\mathcal{B}_2 = (P_2, I_2, U_2)$ эквивалентны. Преобразуем блоки $\mathcal{B}_1$ и $\mathcal{B}_2$ в программы $\mathcal{P}_1$ и $\mathcal{P}_2$. Иными словами, поставим операторы чтения для переменных I_1 и I_2 перед $\mathcal{P}_1$ и $\mathcal{P}_2$ соответственно, а операторы записи для переменных U_1 и U_2 – после P_1 и P_2. Затем к каждой программе присоединим оператор останова. Операторы записи к P_1 и P_2 нужно добавить так, чтобы каждая выходная переменная печаталась хотя бы один раз и последова-

тельности напечатанных значений для $\mathcal{P}_1$ и $\mathcal{P}_2$ были одинаковыми. Поскольку блоки $\mathcal{B}_1$ и $\mathcal{B}_2$ эквивалентны, это всегда можно сделать.

Легко видеть, что программы $\mathcal{P}_1$ и $\mathcal{P}_2$ эквивалентны независимо от того, что именно берется в качестве множества входных присвоения и как интерпретируются функции, представленные операторами входящими в $\mathcal{P}_1$ и $\mathcal{P}_2$. Например, можно взять в качестве входного множества префиксных выражений и считать, что применение операции θ к выражениям $\varepsilon_1, \varepsilon_2, \ldots \varepsilon_r$ дает $\theta\varepsilon_1\varepsilon_2\,\varepsilon_r$.

Однако, если $\boldsymbol{\mathcal{B}}_1$ и $\boldsymbol{\mathcal{B}}_2$ не эквивалентны, но всегда найдется множество типов данных для переменных и интерпретации для операторов, приводящих к тому, что программы $\mathcal{P}_1$ и $\mathcal{P}_2$ будут вырабатывать различные выходные последовательности.

Пример. Рассмотрим программу (алгоритм Евклида). Выходом может быть наибольший общий делитель двух положительных чисел p и q:

```
          read p
          read q
цикл:     r ← remainder(p, q)     (остаток от деления)
          if r = 0 goto выход
          p ← q
          q ← r
          goto цикл
выход:    write q
          halt.
```

Если, например, входным переменным p и q присвоить значения 72 и 76 соответственно, то при обычной интерпретации выходная переменная к моменту выполнения оператора записи будет иметь значение 8. Таким образом, значением этой программы для входного присвоения $p \leftarrow 72$, $q \leftarrow 56$ служит «последовательность», вырабатываемая выходной переменной q.

Если заменить оператор **goto** *цикл* на **if** $q \neq 0$ **goto** *цикл*, то получим эквивалентную программу. Это следует из того, что оператора **goto** *цикл* нельзя достичь, если в четвертом операторе не выполняется условие $r \neq 0$. Поскольку в шестом операторе q принимает значение r, то выполнение седьмого оператора равенство $q=0$ невозможно.

Следует отметить, что преобразования, которые мы можем применять, в значительной степени определяются теми алгебраическими законами, которые мы считаем справедливыми.

Пример. Для некоторых типов данных можно считать, что $a*a=0$, тогда и только тогда, когда $a=0$. Если принять такой закон, то новая эквивалентная программа будет:

```
            read p
            read q
цикл:  r ← remainder(p, q)
            t ← r * r
            if t = 0 goto выход
            p ← q
            q ← r
            goto цикл
выход:  write q
            halt
```

Конечно, при любых мыслимых обстоятельствах эта программа ничем не лучше, но если сформулированный выше закон не верен, то эта программа и первая программа могут оказаться не эквивалентными.

Если дана программа P, то наша цель заключается в нахождении эквивалентной программы P', для которой ожидаемое время выполнения в машинном языке меньше, чем P. Разумным приближением сформулированной задачи будет нахождение такой эквивалентной программы P'', что ожидаемое число машинных команд, меньше числа команд в P.

В большинстве программ одни последовательности операторов исполняются значительно чаще других. Кнут на большом числе программ Фортрана обнаружил, что в типичной программе около половины времени тратится менее чем на 4% программы. Таким образом, практике достаточно применять оптимизирующие процедуры только к многократно проходимым участкам программы. В частности оптимизация может состоять в том, что операторы перемещаются из многократно проходимых областей, редко проходимые, а число операторов в самой программе не меняется или даже не учитывается.

Во многих случаях, можно определить, какой кусок программы будет выполняться чаще других, и вместе с исходной программой передать эту информацию оптимизирующему компилятору. В других случаях довольно просто написать подпрограмму, подсчитывающую, сколько раз выполняется данный оператор. С помощью такого счетчика можно получить «частотный профиль» программы и выяснить те куски, на которые надо направить основные усилия по оптимизации.

6.3.2. Анализ потока управления

Первый шаг на пути оптимизации программ заключается в определении потока управления внутри программы. Для того чтобы сделать это, разобьем программу на группу операторов так, чтобы внутри группу управление передавалось только на первый оператор, и если он выполнился, все остальные операторы группы выполняются последовательно. Такую группу операторов будем называть линейным участком, или просто участком.

Определение. Оператор *S* в программе *P* называется входом в линейный участок, если он

1) первый оператор в *P* или

2) помечен идентификатором, появляющимся после **goto** в операторе перехода либо в условном операторе, или

3) непосредственно следует за условным оператором.

Линейный участок, относящийся к входу в участок S, состоит из *S* и всех операторов , следующих за *S*,

1) вплоть до оператора останова и включая его, или

2) вплоть до входа в следующий линейный участок, но не включая его.

Пример.

```
Участок 1                 read p
                          read q
Участок 2   цикл:r ← remainder(p, q)
                          if r = 0 goto выход
Участок 3                 p ← q
                          q ← r
                          goto цикл
участок 4   выход:    write q
                          halt
```

Из участков программы сконструировать граф, весьма похожий на блок-схему программы.

Определение. Графом управления назовем помеченный ориентированный граф *G*, содержащий выделенную вершину *n*, из которой достижима каждая вершина *G*. Вершину n назовем начальной.

Граф управления программы – это граф управления, в котором каждая вершина соответствует какому-нибудь участку программы. Предположим, что вершины *i* и *j* графа управления соответствуют участкам *i* и *j* программы. Тогда дуга идет из вершины *i* в вершину *j*, если

1) последний оператор участка *i* не является ни оператором перехода, ни оператором перехода, ни оператором останова, а участок *j* следует за участком *i*, или

2) последний оператор участка i является оператором **goto** *L*, где *L* -метка первого оператора участка *j*.

Участок, содержащий первый оператор программы, назовем *начальной вершиной.*

Ясно, что любой участок , недостижимый из начальной вершины, можно удалить из данной программы, не меняя ее значения. Далее будем считать, что все такие участки уже удалены из программы.

Пример. Граф управления программой изображен на рис. 11.14.

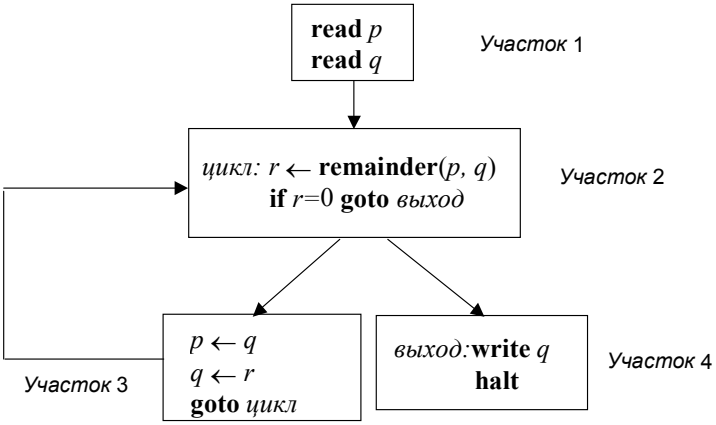


Рис. 11.14. Граф управления.

Многие оптимизирующие преобразования программ требуют знания тех мест в программы, где переменная определяется, и тех, где на ее определение есть ссылка.

Эти связи между определением и ссылкой зависят от последовательности выполняемых участков. Первым участком в последовательности будет начальная вершина, а в каждый должна вести дуга из предыдущего. Иногда предикаты, используемые в условных операторах, могут запрещать переходы по некоторым путям в графе управления. Однако, алгоритма для выявления всех таких операций нет, и мы будем предполагать, что нет «запрещенных путей».

Удобно так же знать, существует ли для участка $\mathcal{B}$, такой участок $\mathcal{B}'$, что всякий раз когда выполняется $\mathcal{B}$, перед ним выполняется $\mathcal{B}'$. В частности, если мы знаем это и если в обоих участках $\mathcal{B}$ и $\mathcal{B}'$ вычисля-

ется одно и то же значение, можно заполнить его после вычисления $\mathcal{B}'$и избежать тем самым перевычисления его в $\mathcal{B}$. Формализуем эти предположения.

Определение. Пусть F-граф управления, имена участков которого выбираются из некоторого множества Δ.

Последовательность участков $\mathcal{B}_1$, $\mathcal{B}_2$, . $\mathcal{B}_n$ из Δ назовем путем вычисления (участков) в F, если

1) $\mathcal{B}_1$– начальная вершина в F,

2) для $1< i \le n$ существует дуга , ведущая из $\mathcal{B}_{i-1}$ в $\mathcal{B}_i$.

Другими словами, путь вычисления $\mathcal{B}_1$, $\mathcal{B}_2$, . $\mathcal{B}_n$ – это путь в F из $\mathcal{B}_1$ в $\mathcal{B}_n$ в котором $\mathcal{B}_1$ – начальная вершина.

Будем говорить, что участок $\mathcal{B}'$ доминирует над участком $\mathcal{B}$, если $\mathcal{B}' \neq \mathcal{B}$ и каждый путь, ведущий из начальной вершины в $\mathcal{B}$, содержит $\mathcal{B}'$.

Будем говорить, что $\mathcal{B}'$ прямо доминирует над $\mathcal{B}$, если

1) $\mathcal{B}'$ доминирует над $\mathcal{B}$ и

2) если $\mathcal{B}''$ доминирует над $\mathcal{B}$ и $\mathcal{B}'' \neq \mathcal{B}'$, то $\mathcal{B}''$ доминирует над $\mathcal{B}'$.

Таким образом, участок $\mathcal{B}'$, прямо доминирует над $\mathcal{B}$, если – «ближайший» к $\mathcal{B}'$ участок, доминирующий над $\mathcal{B}$.

Пример. На рис. ??? – это путь вычисления. Участок 1 прямо доминирует над участком 2 и доминирует над участками 3 и 4. Участок 2 прямо доминирует над участком 3 и 4.

Лемма.

1) Если $\mathcal{B}_1$ доминирует над $\mathcal{B}_2$, а $\mathcal{B}_2$ над $\mathcal{B}_3$, то $\mathcal{B}_1$ доминирует над $\mathcal{B}_3$ (транзитивность).

2) Если $\mathcal{B}_1$ доминирует над $\mathcal{B}_2$, то $\mathcal{B}_2$ не доминирует над $\mathcal{B}_1$ (асимметричность).

3) Если $\mathcal{B}_1$ и $\mathcal{B}_2$ доминируют над $\mathcal{B}_3$, то либо $\mathcal{B}_1$ доминирует над $\mathcal{B}_2$, либо $\mathcal{B}_2$ над $\mathcal{B}_1$.

Лемма. Каждый участок, кроме начальной вершины, имеет единственный прямой доминатор.

Алгоритм вычисления прямого доминирования

Вход. Граф управления F и множество $\Delta = \{\mathcal{B}_1, \mathcal{B}_2, . \mathcal{B}\}$.

Выход. Прямой доминатор DOM($\mathcal{B}$) участка $\mathcal{B}$ для которого $\mathcal{B}$эΔ, кроме начальной вершины.

Метод. DOM($\mathcal{B}$) вычисляется рекурсивно для каждого из $\Delta-\{\mathcal{B}_1\}$. В любой момент DOM($\mathcal{B}$) будет участком ближайшим к $\mathcal{B}$. Среди всех участков, для которых уже известно, что они доминируют над $\mathcal{B}$. В конечном итоге DOM($\mathcal{B}$), будет прямым доминатором участка $\mathcal{B}$. Вначале DOM($\mathcal{B}$) – это $\mathcal{B}_1$ для всех из $\Delta-\{\mathcal{B}_1\}$. Для i =1,2,3..n выполняются следующие два шага:

1) Исключаем участок $\mathcal{B}_i$ из F. Находим все участки $\mathcal{B}$, ставшие теперь недостижимыми из начальной вершины F. Участок $\mathcal{B}_i$ доминирует над $\mathcal{B}$ тогда и только тогда , когда $\mathcal{B}$ становится недостижимым из начальной вершины после исключения $\mathcal{B}_i$ из F. Снова заносим $\mathcal{B}_i$ в F.

2) Предположим, что на шаге 1) обнаружено, что $\mathcal{B}_i$ доминирует над $\mathcal{B}$. Если DOM($\mathcal{B}$)=DOM($\mathcal{B}_i$), берем $\mathcal{B}_i$ в качестве DOM($\mathcal{B}$). В противном случае DOM($\mathcal{B}$) не меняем.

Пример. Применим данный алгоритм к графу управления алгоритма Евклида. Здесь $\Delta=\{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$. Последовательные значения DOM($\mathcal{B}$) после исследований участков $\mathcal{B}_i$ $2\le i \le 4$ приведены в табл. 11.5.

Таблица 11.5.

I	DOM($\mathcal{B}_2$)	DOM($\mathcal{B}_3$)	DOM($\mathcal{B}_4$)
Вначале	$\mathcal{B}_1$	$\mathcal{B}_1$	$\mathcal{B}_1$
2	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_2$
3	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_2$
4	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_2$

Вычислим строку 2. После исключения $\mathcal{B}_2$ участки $\mathcal{B}_3$ и $\mathcal{B}_4$ становятся недостижимые. Таким образом, $\mathcal{B}_2$ доминирует над $\mathcal{B}_3$ и $\mathcal{B}_4$. Перед этим DOM($\mathcal{B}_2$) = DOM($\mathcal{B}_3$) = $\mathcal{B}_1$, тогда в соответствии с шагом 2 выберем $\mathcal{B}_2$ в качестве DOM($\mathcal{B}_3$). Аналогично, полагаем $\mathcal{B}_2$ = DOM($\mathcal{B}_4$). Исключение $\mathcal{B}_3$ и $\mathcal{B}_4$ не делает никакой участок недостижимым.

Отметим, что если F строится из программы, то число дуг не более чем вдвое превосходит число участков. Потому шаг 1) алгоритма выполняется за время пропорциональное квадрату числа участков. Требуется емкость памяти пропорциональная числу участков.

Если $\mathcal{B}_1$, $\mathcal{B}_2$,.., $\mathcal{B}_n$ участок программы, то их доминаторы можно записать как последовательность ε_1, ε_2,.., ε_n, где ε_i прямой доминатор для $\mathcal{B}_i$.

6.3.3. Примеры преобразования программ

Полного каталога оптимизирующих преобразований программ с циклами не существует, однако для широкого класса программ можно использовать следующие преобразования.

Удаления бесполезных операторов

Это - обобщение топологического преобразование T_1. Без оператора, не влияющего на значение программы можно обойтись. Линейные участки, недостижимые из начальной вершины, очевидно, бесполезны, и их можно удалить. Операторы, вычисляющие значения, не используемые в конечном итоге при вычислении выходной переменной, такие попадают в эту категорию.

Исключение избыточных вычислений

Это преобразование обобщает топологическое преобразование T_2. Предположим, что у нас есть программа, в которой участок $\mathcal{B}$ доминирует над $\mathcal{B}'$, и что и содержит операторы $A \leftarrow B+C$ и $A' \leftarrow B+C$. Если B и C не переопределены (выяснить это не трудно), то значения вычисленные этими двумя выражениями совпадают. Тогда в $\mathcal{B}$ после вычисления $A \leftarrow B+C$ можно вставить оператор $X \leftarrow A$, где X- новая переменная. Затем $A' \leftarrow B+C$, можно заменить на $A' \leftarrow X$. Кроме того, если A, нигде на пути из $\mathcal{B}$ в $\mathcal{B}'$ не переопределяется, то оператор $X \leftarrow A$ не нужен, а $A' \leftarrow B+C$ можно заменить $A' \leftarrow A$.

Пример. Рассмотрим граф управления, изображенный на рис. 11.15.

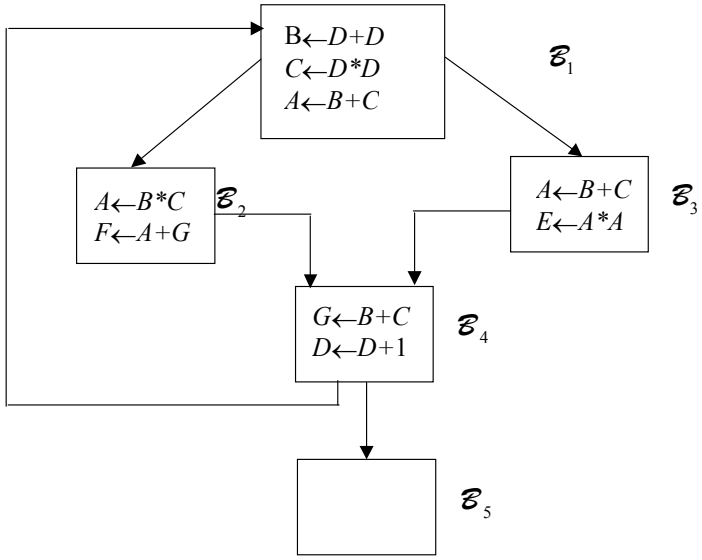


Рис. 11.15. Граф управления.

В этом графе $\mathcal{B}_1$ доминирует над $\mathcal{B}_2$, $\mathcal{B}_3$ и $\mathcal{B}_4$. Тогда $B+C$ принимает одно и то же значение при вычислении в $\mathcal{B}_1$, $\mathcal{B}_3$ и $\mathcal{B}_4$. Поэтому в $\mathcal{B}_3$ и $\mathcal{B}_4$ не обязательно перевычислять выражения $B+C$.

В $\mathcal{B}_1$ после оператора $A\leftarrow B+C$, можно поместить оператор присвоения $X\leftarrow A$. Тогда в $\mathcal{B}_3$ и $\mathcal{B}_4$ операторы $A\leftarrow B+C$ и $G\leftarrow B+C$ можно заменить $A\leftarrow X$ и $G\leftarrow X$ соответственно. Отметим, что A вычисляется в $\mathcal{B}_2$, вместо X нельзя использовать A.

Теперь в $\mathcal{B}_3$ присвоение $A\leftarrow X$ становится избыточным, его можно исключить.

Для исключения из программ избыточных вычислений (общих подвыражений) надо выявить вычисления, общие для двух или более участков программы. Избыточные вычисления, общие для участка и какого-нибудь из доминантов мы рассмотрим. Однако, выражения $A+B$ могут вычисляться в нескольких участках, ни один из которых ни доминирует над данным участком $\mathcal{B}$. Вообще выражение $A+B$ считается избыточным в участке $\mathcal{B}$ если

1) Любой путь из начального участка в $\mathcal{B}$ (в том числе и проходящий через несколько раз) проходит через вычисление $A+B$,

2) Вдоль любого такого пути между последовательным вычислением $A+B$ и использованием $A+B$ в $\mathcal{B}$ не встречается переопределение ни A, ни B.

Отметим, что как и в линейном случае, применение алгебраических законов может увеличить число общих подвыражений.

Замена вычисления периода выполнения вычислениями периода компиляции

Если это возможно, то имеет смысл выполнить вычисление один раз при компиляции, а не повторять его многократно при исполнении объектной программы. Простой пример – *размножение констант*, т.е. замена переменной на константу, когда значение переменной постоянно и известно.

Пример.

read *R*
PI ← 3.14159
A ← 4/3
B ← *A**PI
C ← *R* ↑3
V ← *B***C*
write *V*

В четвертом операторе вместо PI можно подставить 3.14159 и получить *B* ←*A**3.14159. Можно вычислить 4/3 и, подставив найденные выражения в *B* ← *A**3.14159, получить *B* ← 1.3333*3.14159. Можно вычислить 1.3333*3.14159 = 4.18878 и, подставив его в оператор *V* ← *B***C* , получить *V* ← 4.18849**C*. Наконец можно удалить получившиеся бесполезные операторы. В результате у нас будет более короткая эквивалентная программа:

read *R*
C←*R* ↑ 3
V ← 4.18878**C*
write *V*.

Замена сложных операций

Замена сложных операций представляет собой замещение одной операции, занимающей довольно много машинного времени, более быстрой последовательностью.

Пример.

I = LENGTH($S1$ || $S2$)

где $S1$ и $S2$ цепочки переменной длины, а || означает конкатенацию. Реализовать конкатенацию цепочек довольно сложно. Предположим, что мы заменяем этот оператор эквивалентным оператором

I = LENGTH($S1$) + LENGTH($S2$).

Теперь мы дважды выполняем операцию определения длины и один раз сложения. Но эти операции занимают существенно меньше времени, как и конкатенация цепочек.

Другие примеры оптимизации такого типа: замена некоторых умножений сложениями и замена некоторых возведения в степень умножениями. Например C $\leftarrow R\uparrow 3$ можно заменить последовательностью

$C \leftarrow R*R$

$C \leftarrow C*R$

это дешевле, чем вызвать подпрограмму вычисления R^3 как ANTILOG(3*LOG(R))

6.3.4. Оптимизация циклов

Цикл в программе - это последовательность участков, которая может выполняться повторно. Часто сложно добиться существенных улучшений в смысле времени выполнения программы, применяя преобразования уменьшающие оценку циклов. Универсальные преобразования, которые мы рассматривали, в именно, удаление бесполезных операторов, исключение избыточных вычислений, размножение констант, замена сложных вычислений, полезны и в применении к циклам. Существует, однако, и другие преобразования, ориентированные специально на циклы. Это вынесение вычислений из циклов, замена дорогих операций в цикле более дешевыми и развертывание циклов.

Для того, чтобы применить эти преобразования, цикл сначала надо выделить из данной программы. В случае цикла DO в Фортране, или промежуточного кода образуемого циклом DO, найти цикл просто. Однако понятие цикла в графе управления более обще. Эти обобщенные циклы в графе называют сильно «связанными областями».

Любой цикл графа управления с единственной точкой входа служит примером сильно связанной области. Однако, и более общие структуры циклов также служат примерами сильно связанных областей.

Определение. Пусть *F*- граф управления, а $\mathcal{Y}$-подмножество его участков. Будем называть $\mathcal{Y}$ сильно связанной областью в *F*, если

1) в $\mathcal{Y}$ существует единственный участок $\mathcal{B}$ (вход), что найдется путь из начальной вершины графа *F* в $\mathcal{B}$, не проходящий ни через какой другой участок из $\mathcal{Y}$

2) существует путь, не нулевой длины лежащий целиком в $\mathcal{Y}$ и ведущий из участка $\mathcal{Y}$ в любой другой участок в $\mathcal{Y}$.

Пример. Рассмотрим абстрактный граф управления рис. 11.16.

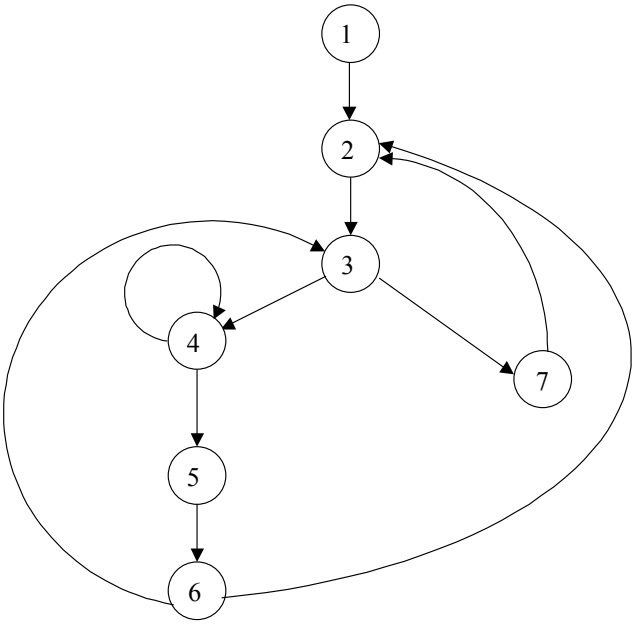


Рис. 11.16. Граф управления.

{2,3,4,5}- сильно связанная область с входом 2
{4}- сильно связанная область с входом 4
{3,4,5,6}- область с входом 3
{2,3,7}- область с входом 2
{2,3,4,5,6,7}- область с входом 2

Последняя максимальна в том смысле, что любая другая область с входом 2 содержится внутри этой области.

Важной особенностью сильно связанной области, благодаря которой она помогает улучшить код, является однозначность определения входного участка.

Теорема. Пусть F-граф управления. Участок $\mathcal{B}$ в F является входным участком области тогда и только тогда, когда существует такой участок $\mathcal{B}'$, что из него есть дуга в $\mathcal{B}$ и $\mathcal{B}$ либо доминирует над $\mathcal{B}'$, либо совпадает с $\mathcal{B}'$.

Перемещение кода

Существует несколько преобразований, в которых для улучшения кода можно воспользоваться знанием областей. Одно из важнейших – *перемещение кода.* Вычисления, не зависящие от области, можно вынести за ее пределы. Пусть внутри некоторой области с одним входом переменные Y и Z не меняются, но есть оператор $X \leftarrow Y+Z$. Вычисление $Y+Z$ можно переместить в заново образованный участок области, связанный только с входным участком. Все связи вне области, раннее вошедшие во входной участок, теперь идут в новый участок.

Пример.

```
          K=0
          DO 3 I=1,1000
3         K=J+1+I+K
```

Промежуточная программа для этого фрагмента исходной программы может быть такой.

```
          K← 0
          I ← 1
цикл:  T ← J + 1
       S ← T + I
          K← S + K
          if I = 1000 goto выход
          I ← I+1
          goto цикл
выход:    halt
```

Граф этой программы приведен на рис. 11.17.

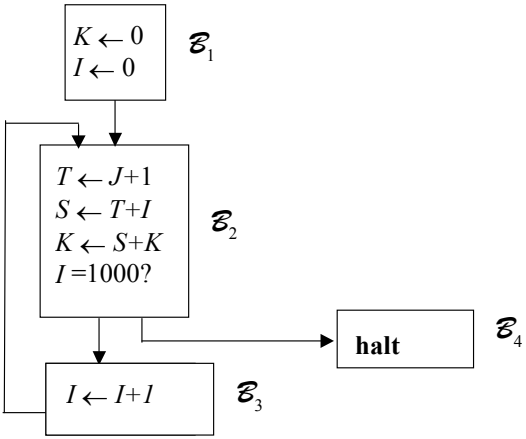


Рис. 11.17. Граф управления.

Из графа видно, что $\{\mathcal{B}_2, \mathcal{B}_3\}$ область с входом 2. Оператор $T \leftarrow J+1$ инвариантен в этой области, так, что его можно перенести на новый участок, как показано на рис. 11.18.

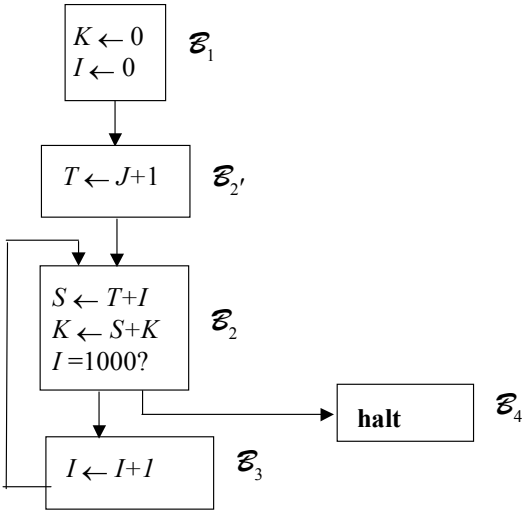


Рис. 11.18. Преобразованный граф управления.

Несмотря на то, что в новом графе (программе) столько же операторов, что и в предыдущем, операторы в области будут выполняться часто, так что ожидаемое время выполнения уменьшится.

Индуктивное перемещение

Определение. Пусть $\mathcal{Y}$ область определения с одним входом, а X переменная, появляющаяся в некотором операторе, входящем в один из участков $\mathcal{Y}$. Пусть $\mathcal{B}_1\mathcal{B}_2$ .. $\mathcal{B}_n\varepsilon_1,\varepsilon_2,..,\varepsilon_m$ – такой путь вычисления, что ε_i принадлежит $\mathcal{Y}$, $1 \le i \le m$. Обозначим через X_1, X_2,… значения, присеваемые X в последовательности $\mathcal{B}\varepsilon_1\varepsilon_2$.. ε_m. Если X_1, X_2.. образуют арифметическую прогрессию (с положительной или отрицательной разностью) для любого пути вычисления типа указанного выше, то будем X называть *индуцированной* переменной в $\mathcal{Y}$.

Будем также называть X индуцированной переменной, если она в $\mathcal{B}$ неопределена и ее значения образуют арифметическую прогрессию.

Отметим, что задача нахождения всех индуцированных переменных в области нетривиальна (общего алгоритма не существует, но для частных случаев решения достаточно просты).

Пример. На рис.11.18 области {$\mathcal{B}_2$ и $\mathcal{B}_3$} имеют вход $\mathcal{B}_2$. Если вход в $\mathcal{B}_2$ осуществляется из $\mathcal{B}_2'$ и управление повторно передается от $\mathcal{B}_2$ к $\mathcal{B}_3$ и снова к $\mathcal{B}_2$, то переменная I принимает значения 1,2,3,.. Таким образом, I – индуцированная переменная. Менее очевидно, что S – так же индуцированная переменная, поскольку она принимает значения T+1, T+2,. T+3, а поскольку K не индуцированная, то она также принимает значения T+1, 2T+3, 3T+6 ..

Важна особенность индуцированных переменных – их линейная связь друг с другом при передаче управления внутри области, которой они принадлежат. Например, для рис. 11.18 каждый раз при выходе из $\mathcal{B}_2$, справедливы соотношения S=T+1 и I=S-T.

Если, как на рис. 11.18 какая-то индуцированная переменная используется только для управления в области (на это указывает тот факт, что ее значение не требуется за пределами области и что непосредственно перед входом в область ей присваивается всегда одна и та же константа), то ее можно исключить. Даже если за пределами области требуются все индуцированные переменные, внутри области можно использовать одну, а все остальные вычислить при выходе из области.

Пример. Рассмотрим рис. 11.18. Исключим индуцированную переменную *I*. Ее роль будет играть *S*. Заметим, что после участка $\mathcal{B}_2$, переменная *S* принимает значение *T*+*I*, так, что когда управление возвращается от $\mathcal{B}_3$ к $\mathcal{B}_2$ должно выполняться соотношение *S*=*T*+*I*-1. Таким образом, оператор $S \leftarrow T+I$, можно заменить на $S \leftarrow S+1$. Но затем в $\mathcal{B}_2'$ надо правильно инициировать *S*, так что, когда управление перейдет из $\mathcal{B}_2'$ в $\mathcal{B}_2$ значение *S*, после оператора будет *T*+1.

Затем мы должны исправить проверку *I*=1000?, так, чтобы получить эквивалентную проверку относительно *S*. При выполнении этой проверки *S* имеет значение *T*+*I*. Следовательно, эквивалентной проверкой будет

$R \leftarrow T+1000$
$S=R?$

Поскольку *R* не зависит от области, вычисление $R \leftarrow T+1000$ можно вынести в участок $\mathcal{B}_2'$. Тогда можно полностью избавиться от *I*. Новый граф представлен на рис. 11.19.

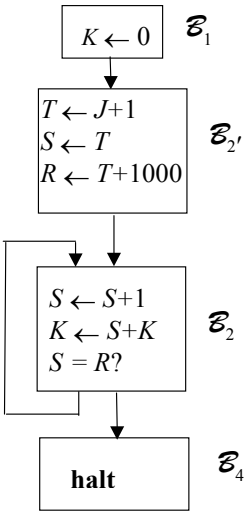


Рис. 11.19. Дальнейшее преобразование графа управления.

На рис11.19 видно, что участок $\mathcal{B}_3$ исключен полностью, а область укорочена на один оператор. Конечно, размер участка $\mathcal{B}_2'$ увеличился,

но по-видимому, области исполняются значительно чаще, чем участки вне области. Т.е. имеем ускоренный вариант программы.

Шаг $S \leftarrow T$ в $\mathcal{B}_2'$ можно исключить, если отождествить S и T. Это возможно только по тому, что значения переменных S и T никогда не будут различными, но не смотря на это обе они будут «активными» в том смысле, что будут использоваться в дальнейших вычислениях. Иными словами в $\mathcal{B}_2$ активна только переменная S, ни одна из них не активна в $\mathcal{B}_1$, а в $\mathcal{B}_2'$ обе активны между операторами $S \leftarrow T$ и $R \leftarrow T+1000$. Если T заменить на S получим граф управления рис. 11.20.

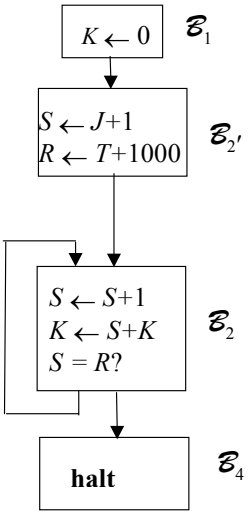


Рис. 11.20. Окончательный вариант графа управления.

Для того чтобы понять, чем результирующий граф лучше исходного, превратим каждый из них в программу на языке ассемблера и введем новые коды операций (JZERO - переход в случае нулевого сумматора и JNZ - переход в случае ненулевого сумматора)

```
LOAD = 0                    LOAD = 0
STORE  K                    STORE  K
LOAD = 1                    LOAD = J
цикл:       STORE  I                        ADD  = 1
            LOAD   J                        STORE  S
            ADD  = 1                        ADD  =1000
            ADD    I                        STORE  R
```

```
            ADD    K                цикл:  LOAD   S
            STORE  K                       ADD    = 1
            LOAD   I                       STORE  S
            SUBTR  =1000                   ADD    K
            JZERO  выход                   STORE  K
            LOAD   I                       LOAD   S
            ADD    = 1                     SUBTR  R
            JMP    цикл                    JNZ    цикл
выход:      END                            END
```

а *б*

Заметим, что длина программы такая же, однако цикл короче (8 команд вместо 12).

Замена сложных операций

Внутри областей возможна замена сложных операций. Если внутри области есть оператор вида $A \leftarrow B*I$, в котором значение B не зависит от области, а I индуктивная переменная, то можно заменить умножение сложением или вычитанием величины, равной произведению значений, не зависящих от области разности арифметических программ, порождаемой индуктивной переменной.

Пример.

```
            DO 5 J = 1, N
            DO 5 I = 1, M
5           A(I, J) = B(I, J)
```

Пусть $A(I, J)$ запоминается в ячейке $A+M*(J-1)+I$ для $1 \le I \le M$, $1 \le J \le N$; аналогичное предположение сделаем относительно $B(I, J)$. Для удобства обозначим ячейку $A+L$ через $A(L)$. Тогда из исходной программы можно получить частично оптимизированную программу

```
            N' ← N-1
            J ← -1
            J ← J+1
            I ← 0
            K ← M*J
цикл:       I ← I+1
            L ← K+I
            A(L) ←B(L)
            if I < M goto цикл
```

if $J < N'$ **goto** *цикл*
halt

Граф управления новой программы изображен на рис. 11.21. В этом графе $\{\mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$- кисть, в которой переменная M инвариантна, а J-индуктивная переменная, возврастающая на 1. Поэтому оператор $K \leftarrow M*J$ можно заменить на $K \leftarrow K+M$, предварительно присвоив K значение $-M$ вне области.

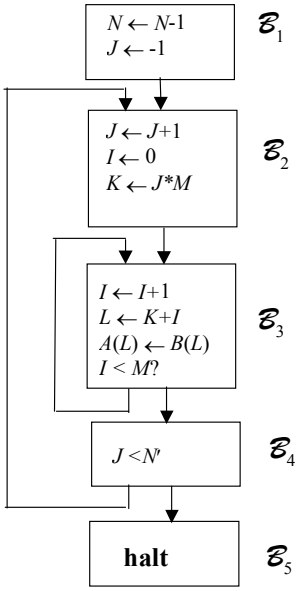


Рис. 11.21. Граф управления.

Новый граф управления представлен на рис. 11.22. Программа представляемая этим новым графом, длиннее прежней, но области, соответствующие участкам $\mathcal{B}_2''$, $\mathcal{B}_3$ и $\mathcal{B}_4$ будут исполняться быстрее, поскольку умножение заменено сложением.

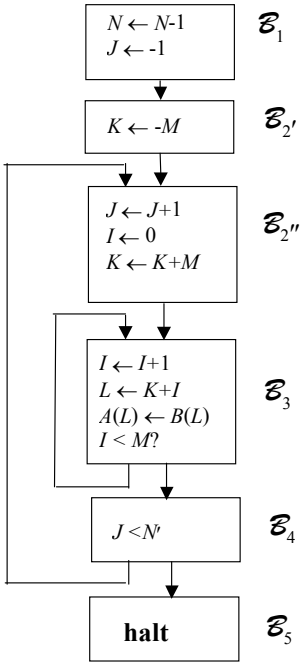


Рис. 11.22. Новый граф управления.

Можно получить более экономную программу, заменив всю область $\{\mathcal{B}_2'', \mathcal{B}_3, \mathcal{B}_4\}$ одним участком. Окончательный граф управления представлен на рис. 11.23.

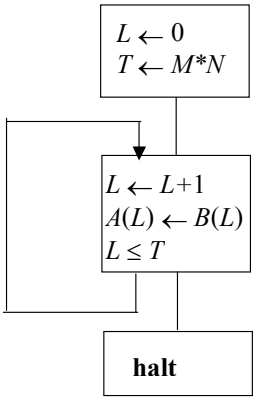


Рис. 11.23. Окончательный граф управления.

Развертывание циклов

Последние преобразование по улучшению кода, которое часто остается незамеченным, это *развертывание циклов.*.

Рассмотрим граф на рис. 11.23.

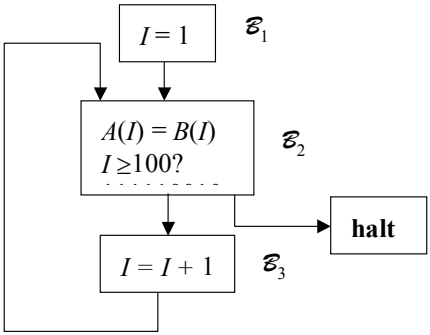


Рис. 11.23. Граф управления.

В этом графе участки $\mathcal{B}_1$ и $\mathcal{B}_2$ выполняются 100 раз. Таким образом, 100 раз выполняется проверка. Не допуская никаких вольностей можно развернуть цикл на «один шаг» и получить граф рис. 11.24.

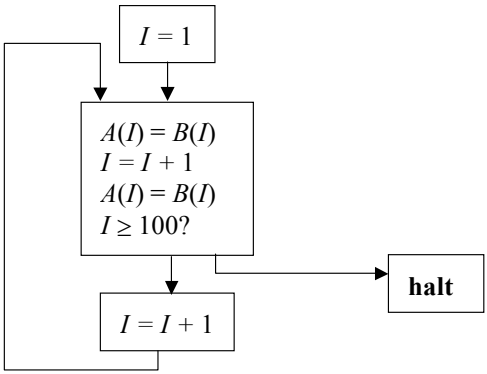


Рис. 11.24. Граф управления после развертывания цикла.

Программа представленная на рис. 11.24 длиннее, но в ней исполняется меньше команд (всего 50 проверок вместо 100)

6.4. Анализ потоков данных

До сих пор мы использовали информацию о вычислениях в участках программ, не описывая, как этот участок и информацию о нем можно получить. В частности мы использовали:

1) «Допустимые» при входе в участок выражения. Выражение $A+B$ называется *доступным* при входе в участок, если $A+B$ всегда вычисляется до достижения участка, но не ранее чем определены A и B.

2) Множество участков, в которых переменная могла определяться последний раз перед тем, как поток управления достиг текущего участка. Эта информация полезна для размножения констант и выявления бесполезных вычисления. Она используется также для выявления возможных ошибок программиста, заключающихся в том, что на переменную делается ссылка до того, как она определена.

Информация третьего типа, для вычисления связана с выявлением активных переменных, то есть переменных, значения которых должны сохраняться при выходе из участка. Эта информация полезна, когда участки преобразуются в машинные коды, поскольку она указывает переменные, которые при выходе из участка должны либо запоминаться, либо сохраняться в быстром регистре (то есть та информация нужна для выявления выходных переменных). Отметим, что переменная может вычисляться не в рассматриваемом участке, а в каком либо

предыдущем, но быть, тем не менее, входной и выходной переменной участка.

Самая сложная из этих проблем – вторая – установление участка, где могла определяться переменная перед тем, как был, достигнут данный участок. Метод решения этой проблемы оказывается «анализом интервалов». Он заключается в разбиение графа управления на все большее и большее множества вершин; тем самым с графом связывается некоторая иерархическая структура. С помощью этой структуры можно будет дать эффективный алгоритм для класса графов управления, называемых «сводимыми», такие графы очень часто встречаются в качестве графов управления, возникающих из реальных программ.

6.4.1. Интервалы

Определение. Если h – вершина графа управления F, определим интервал $I(h)$ с *заголовком* h как такое множество вершин графа F, что

1) h принадлежит $I(h)$,

2) если вершина n, еще не включена в $I(h)$ и все дуги входящие в n, выходят из вершин, принадлежащих $I(h)$, добавим n к $I(h)$,

3) Повторяем шаг 2) до тех пор, пока не останется вершин, которые можно добавить к $I(h)$.

Пример. Рассмотрим граф управления, изображенный на рис. 11.25.

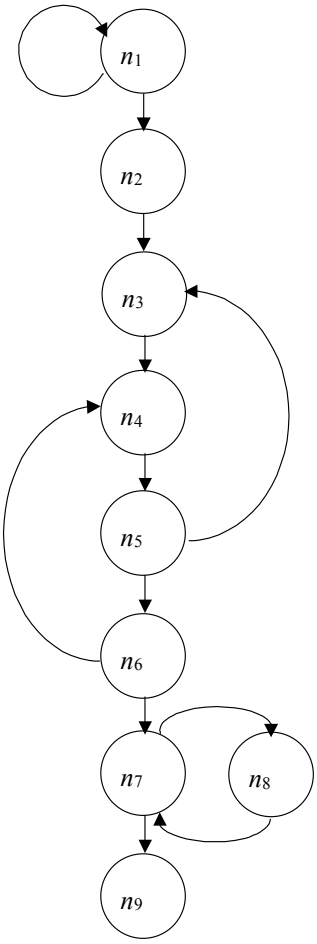


Рис. 11.25. Граф управления.

Рассмотрим интервал с начальной вершиной n_1, в качестве заголовка. Согласно шагу 1) $I(n_1)$ включает n_1. Поскольку единственная дуга, входящая в n_2 выходит из n_1 добавим n_2 к $I(n1)$, вершину n_3 нельзя добавить к $I(n_1)$, так как в нее можно попасть не только из n_2, но и из n_5. Никаких других вершин к $I(n_1)$ добавить нельзя. Таким образом, $I(n_1) = \{n_1, n_2\}$

Продолжая разбор, получим:

$I(n_1) = \{n_1, n_2\}$
$I(n_3) = \{n_3\}$
$I(n_4) = \{n_4, n_5, n_6\}$
$I(n_7) = \{n_7, n_8, n_9\}$

Остается проблема нахождения заголовков.

Теорема.

1) Заголовок h доминирует над всеми остальными вершинами в $I(h)$, хотя не обязательно все вершины над которыми он доминирует принадлежат $I(h)$.

2) Для каждой вершины h графа управления F интервал $I(h)$ определяется однозначно и не зависит от порядка, в котором на шаге 2) определения интервала выбираются кандидаты для n.

3) Каждый цикл в интервале $I(h)$ включает заголовок интервала h

Важным следствием этой теоремы является факт, что графы управления можно единственным образом разбить на интервалы, а интервалы одного графа управления, в котором из интервала I_1 ведет дуга в другой интервал I_2, если какая-нибудь дуга ведет из вершины интервала I_1 в заголовок интервала I_2. (Ясно, что никакая дуга не может вести

из I_1 в вершину интервала I_2, отличную от заголовка). Новый граф можно таким же образом разбить на интервалы, и этот процесс можно продолжить. Поэтому в дальнейшем мы будем считать, что граф управления состоит не из участков, а из вершин, тип которых не специфицирован. Иначе, вершины могут представлять структуры произвольной сложности.

Алгоритм разбиения графа управления на непересекающиеся интервалы.

Вход. Граф управления F.

Выход. Множество непересекающихся интервалов, объединение которых содержит все вершины графа F.

Метод.

1) С каждой вершиной в F свяжем два параметра: счетчик и достижимость. Счетчик для n сначала равен числу дуг входящих в n. В ходе выполнения алгоритма счетчик для n равен числу еще не пройденных дуг, входящих в n. Достижимость для n либо не определена, либо является некоторой вершиной из F. Вначале достижимость не определена для всех вершин, кроме начальной, достижимость которой есть она сама. В конечном итоге достижимостью для n станет первый найденный заголовок интервала n, такой, что из некоторой вершины интервала $I(n)$ ведет дуга в n.

2) Образуем список вершин, называемым *списком заголовков*. Вначале список заголовков содержит только начальную вершину графа F.

3) Если список заголовков пуст, остановиться. В противном случае n – следующая вершина из списка заголовков.

4) Затем применяем шаги 5)- 7) для построения интервала $I(n)$. На этих шагах к списку заголовков добавляются прямые потомки вершины из $I(n)$.

5) $I(n)$ строим как список вершин. Вначале $I(n)$ содержит только вершину n и она «не помечена».

6) Выбираем в $I(n)$ «непомеченную вершину» n', помечаем ее и для каждой вершины n'', в которую ведет дуга из n', выполняем следующие операции:

(а) Уменьшаем счетчик на 1 для n''

(б) (i) Если достижимость для n'' не определена, полагаем ее равной n и делаем следующее. Если счетчик для n'' равен 0 (перед этим был 1) то добавляем вершину n''' к $I(n)$ и переходим к шагу 7); иначе добавляем вершину n' к списку заголовков, если ее там не было, и переходим к шагу 7).

(ii) Если достижимость для n'' равна n, а счетчик вершин n'' равен 0, добавляем вершину n'' к $I(n)$ и удаляем ее из списка заголовков, если она там есть и переходим к шагу 7.

Если ни 1) ни 2) не применимы, в б) ничего не делаем.

7) Если в $I(n)$ остается непомеченная вершина, возвращаемся к шагу 6). Иначе список $I(n)$ заполнен, возвращаемся к шагу 3).

Определение. Из интервалов графа управления F, можно построить другой граф управления $I(F)$, который будем называть производным графом от F. Произвольный граф определяется так:

1) $I(F)$ имеет по одной вершине для каждого интервала, построенного ниже описанным алгоритмом

2) Начальной вершиной I(F) служит интервал, содержащей начальную вершину для F.

3) Из интервала I в интервал J ведет дуга тогда и только тогда, когда $I \neq J$ и из вершины I ведет дуга в заголовок интервала J.

Произвольный граф $I(F)$ графа управления F показывает поток управления между интервалами в F. Поскольку граф $I(F)$ сам является графом управления можно построить также граф $I(I(F))$ производный от $I(F)$. Таким образом, если дан граф управления F_0, можно построить последовательность графов управления F_0, F_1,..., F_n, называемую *производной последовательностью* от F, в которой F_{i+1} производный граф от F_i. Граф F_i – называется i – производным графом от F_0. Граф F_n – называется пределом графа F_0. Нетрудно показать, что F_n всегда существует и единственен.

Если F_n состоит из одной вершины, то граф F_0 называется *сводимым*.

Следует отметить, что если граф F_0 строится по реальной программе, то он всегда сводимый.

Пример. Применим рассмотренный алгоритм для построения интервалов управления для граф рис. 11.26.

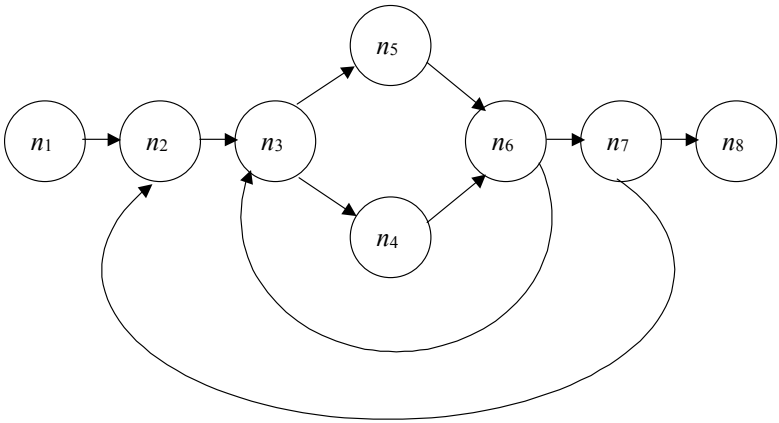


Рис. 11.26. Граф управления.

Начальной вершиной служит n_1. Вначале список заголовков содержит только n_1. Для построения $I(n_1)$ включаем n_1 в $I(n_1)$ как непомеченную вершину. Помечаем вершину n_1 ее прямым потомком n_2. Для этого уменьшаем счетчик n_2 с 2 до 1, полагаем достижимость до нее $n1$ и добавляем ее к списку заголовков. К этому моменту $I(n_1)$ не остается непомеченных вершин, так что список $I(n_1) = \{n_1\}$ заполнен.

Список заголовков содержит потомка n_2 вершины из $I(n_1)$. Для вычисления $I(n_2)$ включаем n_2 в $I(n_2)$ и рассматриваем вершину n_3, счетчик для которой равен 2. Уменьшаем счетчик на 1, полагаем достижимость для нее равной n_2 и добавляем ее к списку заголовков. Находим таким образом, что $I(n_2) = \{n_2\}$.

Список заголовков содержит теперь потомка n_3 из $I(n_2)$. Вычисление $I(n_3)$ начинаем теперь с занесения n_3 в $I(n_3)$. Рассматриваем теперь вершины n_4 и n_5, уменьшая счетчик для них 1 до 0, полагая достижимость для них равной n_3 и добавляя их к $I(n_3)$ как непомеченные вершины. Помечаем n_4, уменьшая счетчик для n_6 с 2 до 1, полагая достижимость для n_6 равной n_3 и добавляя n_6 к списку заголовков. Помечая n_5 в $I(n_3)$, уменьшаем счетчик для n_6 с 1 до 0, удаляем ее из списка и добавляем к $I(n_3)$.

Чтобы пометить n_6 в $I(n_3)$ делаем счетчик для n_7 равным 0, полагаем достижимость для нее равной n_3 и добавляем ее к $I(n_3)$. Следующей рассматривается вершина n_3, поскольку есть дуга из n_6 в n_3. Так как достижимость для n_3 равна n_2, вершина n_3 в данный момент не изменяет $I(n_3)$ и списка заголовков. Чтобы пометить n_7, делаем счетчик n_8

равным 0, полагаем достижимость для нее равной n_3 и добавляем ее к $I(n_3)$. Вершина n_2 также является потомком вершины n_7, но так как достижимость для n_2 равна n_1, то n_2 не добавляется ни к $I(n_3)$ ни к списку заголовков.

Наконец чтобы пометить n_8, не надо производить никаких операций, поскольку n_8 не имеет потомков. К этому моменту в $I(n_3)$ не остается непомеченных вершин, так что $I(n_3) = \{n_3, n_4, n_5, n_6, n_7, n_8\}$.

Список заголовков пуст, так что алгоритм закончился. В результате граф управления оказался разбитым на непересекающиеся интервала

$I(n_1) = \{n_1\}$

$I(n_2) = \{n_2\}$

$I(n_3) = \{n_3, n_4, n_5, n_6, n_7, n_8\}$.

На этих интервалах можно построить последовательность графов управления (рис. 11.27).

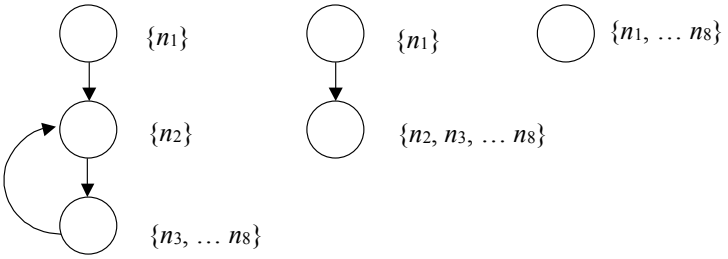


Рис. 11.27. Последовательность графов управления.

Пример. Рассмотрим граф рис. 11.28.

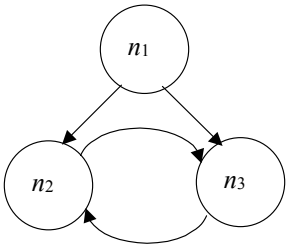


Рис. 11.28. Граф управления *F*.

Интервалы для него таковы:

$I(n_1) = \{n_1\}$
$I(n_2) = \{n_2\}$
$I(n_3) = \{n_3\}$

В соответствие с рассмотренным алгоритмом, находим, что граф F несводим.

Заметим, рассмотренный алгоритм выполняется за время, пропорциональное числу дуг графа управления. Поскольку в графе управления, вершины которого являются участками программы, ни из одной вершины не выходит более 2 дуг, это эквивалентно тому, что алгоритм линейно зависит от участков программы.

6.4.2. Анализ потоков данных с помощью интервалов

Проблема, которую мы будем изучать, состоит в том, что для каждого участка $\mathcal{B}$ и для каждой переменной A сводимого графа управления выяснить в каких операторах программы могла определяться переменная A в последний раз, перед тем как управление достигло участка $\mathcal{B}$.

Будем исходить из априорного утверждения, что анализ потоков данных с помощью интервалов связан с трактовкой графов как упакованных векторных битов. Для вычисления пересечения, объединения и дополнения множеств используются логические операции AND, OR и NOT на векторах битов.

Построим таблицы, дающие для каждого участка $\mathcal{B}$ программы все позиции l, где определяется данная переменная A и откуда существует путь в $\mathcal{B}$, вдоль которого A не переопределяется. Эта информация необходима для вычисления возможных значений A при входе в участок $\mathcal{B}$.

Определим *четыре* функции, отображающие участки во множества.

Определение. Путем вычислений из оператора s_1 в оператор s_2 назовем последовательность операторов, начинающуюся в s_1 и заканчивающуюся в s_2, которая в данном порядке может выполняться в процессе выполнения программы.

Пусть $\mathcal{B}$ участок программы P. Определим четыре множества, связанные с операторами определения:

1) $\mathrm{IN}(\mathcal{B}) = \{d \in P \mid$ существует такой путь вычисления из оператора определения d в первый оператор в $\mathcal{B}$, что никакой оператор на

этом пути, кроме может быть первого оператора в $\mathcal{B}$, не переопределяет переменную, определенную в d}.

2) OUT($\mathcal{B}$) = {$d \in P$ | существует такой путь вычисления из d в последний оператор $\mathcal{B}$, что никакой оператор на нем не переопределяет переменную, определенную в d}.

3) TRANS($\mathcal{B}$) = {$d \in P$ | переменная, определенная в d, не определяется никаким оператором в $\mathcal{B}$}.

4) GEN($\mathcal{B}$) = {$d \in P$ | переменная, определенная в d, не определяется никаким оператором в $\mathcal{B}$}.

Таким образом, IN($\mathcal{B}$) содержит определения, которые могут быть активными при входе в $\mathcal{B}$, OUT($\mathcal{B}$) содержит определения, которые могут быть активными при выходе из $\mathcal{B}$, TRANS($\mathcal{B}$) содержит определения, передаваемые через $\mathcal{B}$ без определения в $\mathcal{B}$, GEN($\mathcal{B}$) содержит определения, создаваемые в $\mathcal{B}$, которые остаются активными при выходе из $\mathcal{B}$.

Легко видеть, что

OUT($\mathcal{B}$) = (IN($\mathcal{B}$) $\cap$ TRANS($\mathcal{B}$)) $\cup$ GEN($\mathcal{B}$).

Пример. Рассмотрим программу

```
S1:  I ← 1
S2:  J ← 0
S3:  J ← J + 1
S4:  read I
S5:  if I < 100 goto S8
S6:  write J
S7:  halt
S8:  I ← I*I
S9:  goto S3
```

Граф программы изображен на рис. 11.29.

Определим для $\mathcal{B}_2$ множества IN, OUT, TRANS и GEN. Оператор $S1$ определяет I, а $S1$, $S2$, $S3$ – путь вычислений, на котором I не определяется (кроме как в $S1$). Поскольку этот путь ведет из $S1$ в первый оператор участка $\mathcal{B}_2$, ясно, что $S1 \in$ IN($\mathcal{B}_2$). Аналогично можно показать, что

IN($\mathcal{B}_3$) = {$S1$, $S2$, $S3$, $S8$}

Отметим, что $S4$ не принадлежит $\mathrm{IN}(\mathcal{B}_2)$, поскольку нет пути вычисления из $S4$ в $S3$ не переопределяющего I после $S4$.

$\mathrm{OUT}(\mathcal{B}_2)$ не содержит $S1$, поскольку все пути вычисления из $S1$ в $S5$ переопределяют I. Далее легко проверить, что

$\mathrm{OUT}(\mathcal{B}_2) = \{ S3, S4\}$

$\mathrm{TRANS}(\mathcal{B}_2) = \varnothing$

$\mathrm{GEN}(\mathcal{B}_2) = \{ S3, S4\}$

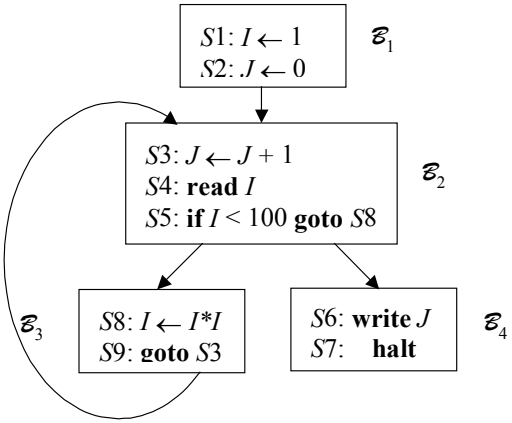


Рис. 11.29. Граф управления.

Предположим, что $\mathcal{B}_1, \ldots, \mathcal{B}_k$ – все прямые потомки участка $\mathcal{B}$ в P. Ясно, что

$$\mathrm{IN}(\mathcal{B}) = \bigcup_{l=1}^{k} \mathrm{OUT}(\mathcal{B}_i) = \bigcup_{l=1}^{k} [(\mathrm{IN}(\mathcal{B}_i) \cap \mathrm{TRANS}(\mathcal{B}_i)) \cup \mathrm{GEN}(\mathcal{B}_i)].$$

Для вычисления $\mathrm{IN}(\mathcal{B})$ можно было бы выписать это уравнение для каждого участка программы вместе с уравнением $\mathrm{IN}(\mathcal{B}_0) = \varnothing$, где $\mathcal{B}_0$ - начальный участок, затем попытаться разрешить систему уравнений. Однако существует более удобный метод, учитывающий преимущества представления графов управления в виде интервалов.

Дадим вначале определения понятий «вход» и «выход» интервала.

Определение. Пусть P – программа, а F_0 ее граф управления. Пусть $F_0, F_1, \ldots, F_n$ – производная последовательность от F_0. Каждая вершина в F_i, $i \geq 1$, является интервалом в F_{i-1} и называется *интервалом порядка i.*

Входом интервала порядка 1 служит заголовок интервала. *Вход* интервала порядка $i > 1$ – это вход в заголовок интервала. Таким образом, вход любого интервала – это линейный участок исходной программы *P*.

Выходом интервала $I(n)$ порядка 1 служит такой последний оператор участка $\mathcal{B}$ в $I(n)$, что $\mathcal{B}$ имеет прямого потомка, который является либо заголовком интервала *n*, либо участком вне $I(n)$. *Выход* интервала $I(n)$ порядка $i \geq 1$ – это последний оператор участка $\mathcal{B}$, содержащегося в $I(n)$ и такого, что в F_0 есть дуга, ведущая из $\mathcal{B}$ либо в заголовок интервала *n*, либо в участок вне $I(n)$.

Отметим, что каждый участок имеет один вход и ноль или более выходов.

Пример. Пусть F_0 – граф управления программы рис. 11.29. С помощью алгоритма разбиения графа на непересекающиеся интервалы, построим его разбиение на интервалы

$$I_1 = I(\mathcal{B}_1) = \{\mathcal{B}_1\}$$

$$I_2 = I(\mathcal{B}_2) = \{\mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$$

Из этих интервалов можно построить первый производный граф управления F_1, показанный на рис11.30. Из F_1 можно построить его интервалы

$$I_3 = I(I_1) = \{ I_1, I_2\} = \{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$$

и получить предельный граф управления, также показанный на рис. 11.30.

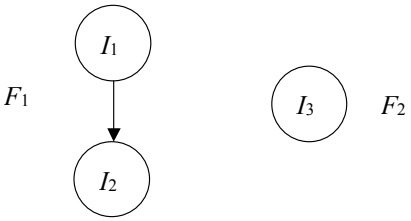


Рис. 11.30. Производная последовательность от графа F_0.

Интервалами порядка 1 являются I_1 и I_2. Вход для I_2 – это $\mathcal{B}_2$. Вход для I_3 – это $\mathcal{B}_1$. Единственный выход для I_1 – это оператор *S*2. Единст-

венный выход для I_2 – это оператор $S9$. Интервалом порядка 2 является I_3 с входом $\mathcal{B}_1$. Интервал I_3 не имеет выходов.

Продолжим теперь функции IN, OUT, TRANS и GEN на интервалы. Пусть F_0, F_1,…, F_n - производная последовательность от F_0, где F – граф управления для P, а I – интервал некоторого графа F_i, $i \geq 1$. Введем следующие основные определения:

1) $\mathrm{IN}(I) = \begin{cases} \mathrm{IN}(\mathcal{B}), \text{ если } I \text{ имеет порядок 1, а } \mathcal{B} \text{ - заголовок для } I, \\ \mathrm{IN}(I'), \text{ если } I \text{ имеет порядок } i > 1, \text{ а } I' \text{ заголовок для } I. \end{cases}$

В 2) – 4) ниже s – вход для I.

2) $\mathrm{OUT}(I, s) = \mathrm{OUT}(\mathcal{B})$, где участок $\mathcal{B} \in I$ таков, что s – его последний оператор.

3) (а) $\mathrm{TRANS}(\mathcal{B}, s) = \mathrm{TRANS}(\mathcal{B})$, если s – последний оператор в $\mathcal{B}$.

(б) $\mathrm{TRANS}(I, s)$ – множество таких операторов $d \in P$, что существует путь без циклов $I_1, I_2,\ldots, I_k$, состоящих исключительно из вершин в I, и такая последовательность выходов $s_1, s_2, \ldots, s_k$ для $I_1, I_2,\ldots, I_k$, соответственно, что

(i) I_1 – заголовок для I,

(ii) в F_0 участок s_j является прямым предком входа для I_{j+1} при $1 \leq j < k$,

(iii) $d \in \mathrm{TRANS}(I_j, s_j)$ при $1 \leq j \leq k$,

(iv) $s_k = s$.

Эти условия иллюстрированы на рис. 11.31.

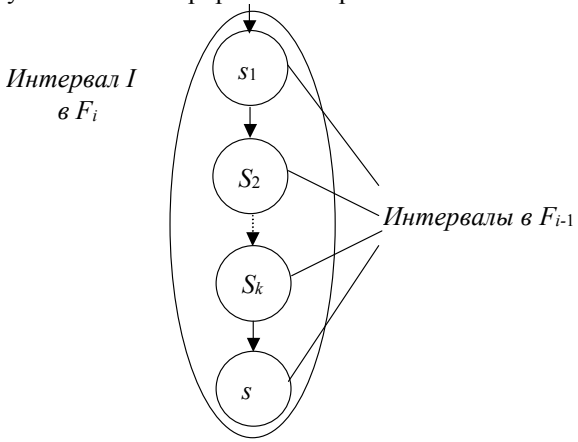


Рис. 11.31. $\mathrm{TRANS}(I_j, s_j)$.

4) (а) GEN($\mathcal{B}$, s) = GEN($\mathcal{B}$), если s - последний оператор в $\mathcal{B}$.

(б) GEN(I, s) - множество таких операторов $d \in P$, что существует путь без циклов $I_1, I_2,\ldots, I_k$, состоящих исключительно из вершин в I, и такая последовательность выходов $s_1, s_2, \ldots, s_k$ для $I_1, I_2,\ldots, I_k$, соответственно, что

(i) $d \in$ GEN(I, s),

(ii) в F_0 участок s_j является прямым предком входа для I_{j+1} при $1 \le j < k$,

(iii) $d \in$ TRANS(I_j, s_j) при $2 \le j \le k$,

(iv) $s_k = s$.

Таким образом, TRANS(I, s_j) – это множество определений, которое можно передать с входа I на выход s без переопределения в I. GEN(I, s) – это множество определений I, которые без переопределений могут достичь s.

Пример. Рассмотрим F_0 на рис 11.29 и F_1 и F_2 на рис. 11.30. В F_1 интервал I_2 это $\{\mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$, и он имеет выход $S9$. Таким образом, IN(I_2) = IN($\mathcal{B}_2$) = {$S1$, $S2$, $S3$, $S8$} и OUT(I_2, s) = OUT($\mathcal{B}_2$) = {$S3$, $S8$}.

TRANS(I_2, $S9$) = $\varnothing$.

GEN(I_2, $S9$) содержит $S8$, поскольку существует последовательность участков, состоящая только из $\mathcal{B}_2$, в которой $S8 \in$ GEN($\mathcal{B}_3$, $S9$). Кроме того, $S3 \in$ GEN(I_2, $S9$) , поскольку существует последовательность участков $\mathcal{B}_2$, $\mathcal{B}_3$ с выходом на $S5$ и $S9$. Это означает, что $S3 \in$ GEN($\mathcal{B}_2$, $S5$), $\mathcal{B}_2$ – прямой предок участка $\mathcal{B}_3$, $S3 \in$ GEN($\mathcal{B}_3$, $S9$).

На основании этих определений дадим алгоритм вычисления IN($\mathcal{B}$) для всех участков программы P. Он обрабатывает программы только со сводимыми графами управления.

Алгоритм вычисления функции IN.

Вход. Сводимый граф F_0 для программы P.

Выход. IN($\mathcal{B}$) для каждого участка $\mathcal{B} \in P$.

Метод.

1) Пусть $F_0, F_1,\ldots, F_k$ - производная последовательность от F_0. Вычислим TRANS($\mathcal{B}$) и GEN($\mathcal{B}$) для всех участков $\mathcal{B}$ из F_0.

2) Для i = 1, 2, …, k последовательно вычисляем TRANS(I, s) и GEN(I, s) для всех интервалов порядка i и выходов s для I. Рекурсивное определение этих функций гарантирует, что это можно сделать.

3) Полагаем IN(I) = $\varnothing$, где I – одиночный интервал порядка k. Устанавливаем i=k.

4) Для всех интервалов порядка i выполняем следующее. Пусть $I = \{I_1, \ldots, I_n\}$ – интервал порядка i ($I_1, \ldots, I_n$ – интервалы порядка i-1 или участки, если i=1). Можно считать, что эти интервалы перечислены в том порядке, в котором из них составлен интервал I в алгоритме определения непересекающихся интервалов. Иными словами, I_1 – это заголовок и для каждого $j > 1$ множество $\{I_1, \ldots, I_{j-1}\}$ содержит все вершины из F_{i-1}, являющиеся прямыми предками интервала I_j.

(а) Пусть $s_1, s_2, \ldots, s_r$ – выходы интервала I, каждый из которых принадлежит участку в F_0, являющемуся прямым предком входа для I. Полагаем

$$\mathrm{IN}(I_1) = \mathrm{IN}(I) \cup \bigcup_{i=1}^{r} \mathrm{GEN}(I, s_i)$$

(б) Для всех $s \in I_1$ полагаем

$$\mathrm{OUT}(I_1, s) = (\mathrm{IN}(I_1) \cap \mathrm{TRANS}(I_1, s) \cup \mathrm{GEN}(I, s))$$

(в) Для j=2, 3, …, n пусть $s_{r1}, s_{r2}, \ldots, s_{rkr}$ – выходы интервала I_r, $1 \le r < j$, каждый из которых принадлежит участку в F_0, являющемуся прямым предком для входа I_r. Полагаем

$$\mathrm{IN}(I_j) = \bigcup_{r,l} \mathrm{OUT}(I_r, s_{rl})$$

$$\mathrm{OUT}(I_j, s) = (\mathrm{IN}(I_j) \cap \mathrm{TRANS}(I_j, s)) \cup \mathrm{GEN}(I_j, s)$$

для всех интервалов, входящих в I_j.

5) Если i = 1 остановиться. В противном случае уменьшить i на 1 и вернуться к шагу 4).

Пример. Применим данный алгоритм к графу управления на рис. 11.29. Для четырех участков в F_0 GEN и TRANS вычисляются просто. Результаты приведены в табл. 11.6.

Таблица 11.6.

Участок	GEN	TRANS
$\mathcal{B}_1$	{*S*1, *S*2}	∅
$\mathcal{B}_2$	{*S*3, *S*4}	∅
$\mathcal{B}_3$	{*S*8}	{*S*2, *S*3}
$\mathcal{B}_4$	∅	{*S*1, *S*2, *S*3, *S*4, *S*8}

Поскольку $\mathcal{B}_3$ определяет только переменную I $\mathcal{B}_3$ «убивает» предыдущие ее определения, но передает определение переменной J, а именно *S*2 и *S*3. Поскольку никакой из участков не определяет переменную дважды, все операторы внутри участка принадлежат множеству GEN для данного участка.

Интервал I_1, состоящий из одного участка $\mathcal{B}_1$, имеет один выход – оператор $S2$. Так как пути в I_1 тривиальны, то $\mathrm{GEN}(I_1, S2) = \{S1, S2\}$ и $\mathrm{TRANS}(I_1, S2) = \varnothing$.

Интервал I_2 имеет один выход $S9$. $\mathrm{GEN}(I_2, S9) = \{S3, S8\}$ и $\mathrm{TRANS}(I_2, S9) = \varnothing$.

Теперь можно начать вычисление функции IN. Первоначально $\mathrm{IN}(I_3) = \varnothing$. Затем к двум подынтервалам в I_3 можно применить шаг 4) алгоритма. Это можно сделать только в порядке I_1, I_2. На шаге 4а) вычисляем $\mathrm{IN}(I_1) = \mathrm{IN}(I_3) = \varnothing$, на шаге 4в) –

$\mathrm{OUT}(I_1, S2) = (\mathrm{IN}(I_1) \cap \mathrm{TRANS}(I_1, S2)) \cup \mathrm{GEN}(I_1, S2) = \{S1, S2\}$.

Далее, на шаге 4в)

$\mathrm{IN}(I_2) = \mathrm{OUT}(I_1, S2) = \{S1, S2\}$.

Проходя по интервалам порядка 1, мы должны рассмотреть составляющие для I_1 и I_2. Интервал I_2 состоит из участков $\mathcal{B}_2$, $\mathcal{B}_3$ и $\mathcal{B}_4$, которые в таком порядке и можно рассматривать. На шаге 4а)

$\mathrm{IN}(\mathcal{B}_2) = \mathrm{IN}(I_2) \cup \mathrm{GEN}(I_1, S9) = \{S1, S2, S3, S8\}$.

На шаге 4б)

$\mathrm{OUT}(\mathcal{B}_2, S5) = \mathrm{IN}(\mathcal{B}_2) \cap \mathrm{TRANS}(\mathcal{B}_2, S5)) \cup \mathrm{GEN}(\mathcal{B}_2, S5) = \{S3, S4\}$.

Поскольку $S5$ ведет к $\mathcal{B}_3$, находим

$\mathrm{IN}(\mathcal{B}_3) = \mathrm{OUT}(\mathcal{B}_2, S5) = \{S3, S4\}$,

а так как $S5$ ведет к $\mathcal{B}_4$, то

$\mathrm{IN}(\mathcal{B}_4) = \mathrm{OUT}(\mathcal{B}_2, S5) = \{S3, S4\}$.

И так,

$$\mathrm{IN}(\mathcal{B}_1) = \varnothing$$

$$\mathrm{IN}(\mathcal{B}_2) = \{S1, S2, S3, S8\}.$$

$$\mathrm{IN}(\mathcal{B}_3) = \{S3, S4\}$$

$$\mathrm{IN}(\mathcal{B}_4) = \{S3, S4\}.$$

Индукцией по порядку интервалов можно доказать, что

1) $\mathrm{TRANS}(I, s)$ – множество таких операторов определений $d \in P$, что существует путь из первого оператора заголовка для I вплоть до s, вдоль которого ни один из операторов не переопределяет переменную, определенную в d,

2) $\mathrm{GEN}(I, s)$ - множество таких операторов определений d, что существует путь из d в s, вдоль которого ни один из операторов не переопределяет переменную, определенную в d.

Индукцией по числу переменных шага 4) можно показать, что

если для вычисления IN(I_j) применяется шаг 4), то IN(I_j) – множество таких определений d, что существует путь из d во вход I_j, вдоль которого ни один из операторов не переопределяет переменную, определенную в d, а OUT(I_j, s) – множество таких d в s, вдоль которого ни один из операторов не переопределяет переменную, определенную в d.

Заключительная теорема. Для всех линейных участков $\mathcal{B} \in P$ множество IN($\mathcal{B}$) – это множество определений d, что в F_0 существует путь из d в первый оператор участка $\mathcal{B}$, вдоль которого ни один из операторов не переопределяет переменную, определенную в d.

6.4.3. Несводимые графы управления

Поскольку не каждый граф сводим, введем еще одно понятие, называемое расщеплением вершин, позволяющее обобщить рассмотренный выше алгоритм на все графы управления. Вершина, в которую входит более одной дуги «расщепляется» на несколько одинаковых копий, по одной на каждую входящую дугу. Таким образом, каждая копия имеет единственную входящую дугу и становится частью интервала для вершины, из которой идет эта дуга. Поэтому расщепление вершин с последующим построением интервала уменьшает число вершин графа по крайней мере на 1.Повторяя этот процесс, можно превратить любой несводимый граф управления в сводимый.

Пример. Рассмотрим несводимый граф управления на рис. 11.28. Вершину n_3 можно расщепить на две копии n_3' и n_3'', получив граф F', изображенный на рис. 11.32.

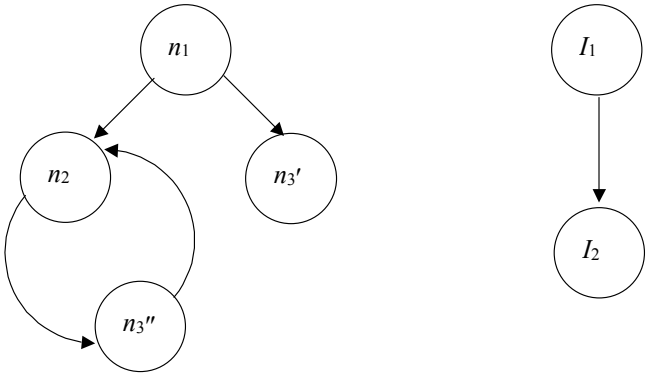


Рис. 11.32. Расщепленный граф управления.

Рис. 11.33. Первый производный граф

Интервалы для графа F' будут

$$I_1 = I_1(n_1) = \{n_1, n_3'\}$$
$$I_2 = I_2(n_2) = \{n_2, n_3''\}$$

Первый производный граф имеет две вершины (рис. 11.33). Второй производный граф состоит из единственной вершины. Таким образом, с помощью расщепления вершин мы превратили граф F в сводимый граф F'.

Дадим модифицированный вариант алгоритма вычисления функции IN, учитывающий этот новый метод.

Лемма. Если граф G – граф управления и $I(G) = G$, то любая вершина n, отличная от начальной, имеет по крайней мере две входящие дуги; ни одна из них не выходит из n.

Доказательство. Все дуги, выходящие из некоторой вершины и входящие в нее же, исчезают при построении интервалов. Поэтому предположим, что вершина n имеет только одну входящую дугу, выходящую из вершины m. Тогда n принадлежит $I(m)$. Если $I(G) = G$, то вершина m в конце концов появится в списке заголовков алгоритма разбиения графа управления на непересекающиеся интервалы. Но тогда n заносится в $I(m)$, так что $I(G)$ не может совпасть с G.

Алгоритм. Общее вычисление функции IN.

Вход. Произвольный граф управления F для программы P.

Выход. IN($\mathcal{B}$) для каждого участка $\mathcal{B} \in P$.

Метод.

1) Для каждого участка $\mathcal{B} \in F$ вычисляем GEN($\mathcal{B}$) и TRANS($\mathcal{B}$). Затем к F рекурсивно применяем шаг 2). Входом для шага 2) служит граф управления G вместе с GEN(I, s) и TRANS(I, s), известными для каждой вершины $I \in G$ и каждого входа s интервала I. Выходом шага 2) служит . IN(I) для каждой вершины $I \in G$.

2)

(а) Пусть G - вход для этого шага и G, G_1,…, G_k – его производная последовательность. Если G_k – одиночная вершина, продолжаем в точности, как и в алгоритме вычисления функции IN. Если G_k – не одиночная вершина, то для всех вершин в G_1,…, G_k можно вычислить GEN и TRANS. Тогда по доказанной лемме G_k содержит некоторую вершину, отличную от начальной, в которую входит более одной дуги. Выберем одну из таких вершин I. Если в I входит j дуг, заменяем I новыми вершинами I_1, I_2, …, I_j. В каждую из I_1, I_2, …, I_j входит по одной

дуге, все они выходят из различных вершин, из которых ране шли дуги в I.

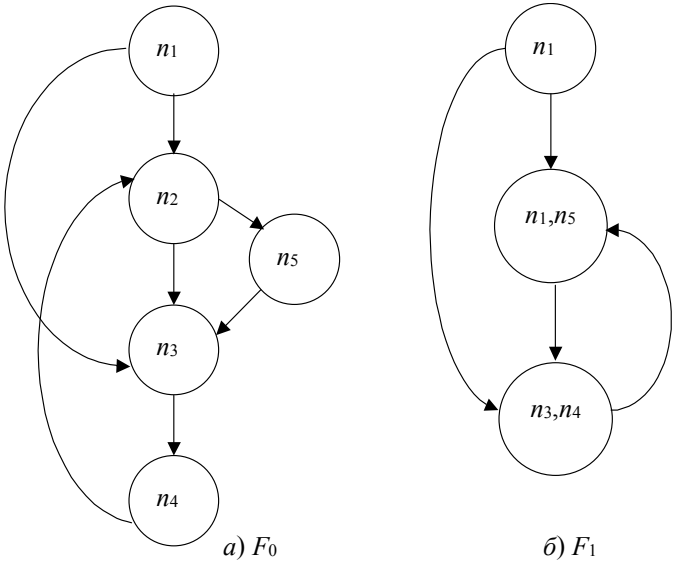
(б) Для каждого выхода s интервала I порождаем выход s_i для $1 \le i \le j$ и считаем, что в F есть дуга, ведущая из каждого s_i во вход каждой вершины, с которой s связаны в G_k. Определяем $\mathrm{GEN}(I_i, s_i) = \mathrm{GEN}(I, s)$ и $\mathrm{TRANS}(I_i, s_i) = \mathrm{TRANS}(I, s)$ для $1 \le i \le j$. Результирующий граф обозначим через G'.

(в) Применим шаг 2) к G'. Функция IN будет рекурсивно вычисляться для G'. Затем полагая $\mathrm{IN}(I) = \bigcup_{i=1}^{j} \mathrm{IN}(I_i)$, вычисляем IN для вершин G_k. Никакие другие изменения для IN не требуются.

(г) Как и в алгоритме вычисляем функции IN для G по IN для G_k.

3) После завершения шага 2) функция IN будет вычислена для каждого участка из F. Эта информация и образует выход алгоритма.

Пример. Рассмотрим граф управления на рис. 11.34.



а) F_0 *б*) F_1

Рис. 11.34. Несводимый граф.

Можно вычислить $F_1 = I(F_0)$, изображенный на рис. 11.34. Однако $I(F_1) = F_1$, так что надо применять процедуру расщепления вершин шага 2). Пусть $\{n_2, n_5\}$ вершина I, которая расщепляется на I_1 и I_2. Ре-

зультат показан на рис. 11.35. Свяжем n_1 с I_1, а $\{n_3, n_4\}$ с I_2. На рисунке изображены дуги из I_1 и I_2 в $\{n_3, n_4\}$. В действительности каждый выход из I дублируется – один для I_1 и один для I_2. С входом $\{n_3, n_4\}$.связаны именно продублированные выходы. Граф на рис. 11.35 сводимый.

В заключении следует отметить, что при построении оптимизирующего компилятора сначала необходимо решить, какие лучше всего применять оптимизации. Решение должно базироваться на характеристиках того класса программ, которые должны компилироваться.

Методы оптимизации арифметических выражений можно использовать при построении окончательной объектной программы. Однако некоторые из них можно ввести в генерацию промежуточного кода, т.е. отдельные части алгоритмов оптимизации арифметических выражений можно встроить во вход синтаксического анализатор.

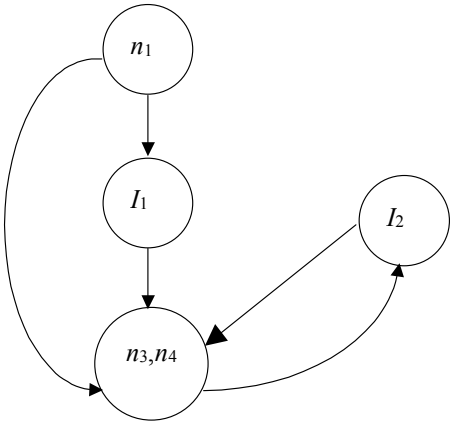


Рис. 11.35. Граф управления.

Это приведет к толу, что на линейных участках программ будут эффективно использоваться регистры.

На фазе компилятора, генерирующего код, имеется программа, которую можно рассматривать как аналог программ с циклами. Здесь основная задача – построить граф управления и оптимизировать циклы, сначала внутренние, затем внешние.

Когда это сделано, можно вычислить глобальную информацию относительно потоков данных. Зная эту информацию можно выполнить «глобальную» оптимизацию – размножение констант и исключение общих выражений.

Наконец, линейные участки можно преобразовать методами оптимизации линейных участков.

7. Включение действий в синтаксис

Синтаксический анализ и генерация кода принципиально различные процессы, но практически во всех компиляторах они выполняются параллельно (т.е. генерация кода осуществляется параллельно с синтаксическим анализом). Наша задача – рассмотреть возможность включения в систему синтаксического анализа действий для генерации кода.

7.1. Получение четверок

В качестве примера включения действия в грамматику для генерирования кода рассмотрим проблему разложения арифметических выражений на четверки. Выражения определяются грамматикой со следующими правилами:

$$S \to EXP$$

$$EXP \to TERM \mid EXP + TERM$$

$$TERM \to FACT \mid TERM \times FACT$$

$$FACT \to -FACT \mid ID \mid (EXP) \mid$$

$$ID \to a \mid b \mid c \mid d \mid e$$

Таким образом, эти правила позволяют анализировать выражения типа:

$$(a + b) \times c$$
$$a \times b + c$$
$$a \times b + c \times d \times e$$

Грамматика для четверок имеет следующие правила:

$QUAD \to OPERAND\ OP1\ OPERAND = INT \mid OP2 = INT$

$OPERAND \to INT \mid ID$

$INT \to DIGIT \mid DIGIT\ INT$

$DIGIT \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$ID \to a \mid b \mid c \mid d \mid e$

$OP1 \to + \mid \times$

$OP2 \to -$

Примеры четверок:

$$-a = 4$$
$$a + b = 7$$
$$6 + 3 = 11$$

Выражение

$$\left(-a + b\right) \times \left(c + d\right)$$

будет соответствовать последовательности четверок:

$$-a = 1$$
$$1 + b = 2$$
$$c + d = 3$$
$$2 \times 3 = 4$$

Целые числа с левой стороны от знаков равенства относятся к другим четверкам. Из сформулированных четверок нетрудно генерировать машинный код, а многие компиляторы на основании четверок осуществляют трансляцию в промежуточный код.

Далее для описания общей схемы разбора примем следующие обозначения: действия заключаются в угловые скобки и обозначаются как *A1, A2*...Для реализации алгоритма четверок требуется четыре действия. Алгоритм пользуется стеком, а номера четверок размещаются с помощью целочисленной переменной. Перечислим эти действия:

A1 – поместить элемент в стек;

A2 – взять три элемента из стека, напечатать их с последующим знаком «=» и номером следующей размещаемой четверки и поместить полученное целое число в стек;

A3 – взять два элемента из стека, напечатать их с последующим значением «=» и номером следующей размещаемой четверки и поместить полученное целое в стек;

A4 – взять из стека один элемент.

Грамматика с учетом этих добавлений примет вид

$$S \to EXP\langle A4\rangle$$
$$EXP \to TERM \mid EXP + \langle A1\rangle TERM\langle A2\rangle$$
$$TERM \to FACT \mid TERM \times \langle A1\rangle FACT\langle A2\rangle$$
$$FACT \to -\langle A1\rangle FACT \mid \langle A3\rangle ID\langle A1\rangle \mid (EXP)$$
$$ID \to a \mid b \mid c \mid d \mid e$$

Действие *A1* используется для помещения в стек всех идентификаторов и операторов, а действия *A2* и *A3* – для получения бинарных и унарных четверок соответственно.

В качестве примера проследим за преобразованием выражения

$$(-a+b)\times(c+d)$$

в четверки. Действие *A1* выполняется после распознавания каждого идентификатора и оператора, действие *A2* – после второго операнда каждого знака бинарной операции, а действие *A3* – после первого (и единственного) оператора каждой унарной операции. Действие *A4* выполняется только один раз после считывания всего выражения.

Последняя считанная литера	Действие	Выход
(	-	
-	*A1*, поместить в стек «-»	
a	*A1*, поместить в стек *a*	
	A3, удалить из стека 2 элемента	*-a=1*
	Поместить в стек «1»	
+	*A1*, поместить в стек «+»	
b	*A1*, поместить в стек *b*	
	A2, удалить из стека 3 элемента	*1+b=2*
	Поместить в стек «2»	
)	-	
×	*A1*, поместить в стек «×»	
(	-	
c	*A1*, поместить в стек *c*	
+	*A1*, поместить в стек «+»	
d	*A1*, поместить в стек *d*	
	A2, удалить из стека 3 элемента	*c+d=3*
	Поместить в стек «3»	
)	-	
	A2, удалить из стека 3 элемента	*2×3=4*
	Поместить в стек «4»	
	A4, удалить из стека 1 элемент	

В рассматриваемом примере нам не пришлось сравнивать приоритеты двух операций, так как эти приоритеты уже заложены в правилах грамматики.

7.2. Работа с таблицей символов

Поскольку синтаксические анализаторы обычно используют контекстно – свободную грамматику, необходимо найти метод определения контекстно – зависимых частей языка. Например, во многих языках идентификаторы не могут применяться, если они ранее не описаны, и имеются ограничения в отношении способов употребления в программе значений различного типа. Кроме того, в языках программирования имеются ограничения на употребление различных знаков. Для запоминания описанных идентификаторов и их типов большинство компиляторов использует таблицу символов. В принятой формализации описания

int *a*

является *определяющей* реализацией *a*, а использование *a* в другом контексте

a=4 или *a+ b* или **read**(*a*)

говорит, что имеется *прикладная* реализации *a*.

Во многих языках программирования один и тот же идентификатор может использоваться для представления в различных частях программы различных объектов (например, в «голове» **int,** а в подпрограмме **char**). В этом случае в таблице символов - это два разных объекта.

Таблица символов имеет ту же блочную структуру, что и сама программа, чтобы различать виды употребления одного и того же идентификатора. При построении таблицы символов учитываются основные свойства большинства языков:

1) определяющая реализация идентификатора появляется раньше любой прикладной реализации;
2) все описания в блоке помещаются раньше всех операторов и предложений;
3) при наличии прикладной реализации идентификатора, соответствующая определяющая реализация находится в наименьшем включающем блоке, в котором содержится описание этого идентификатора;
4) в одном и том же блоке идентификатор не может описываться более одного раза.

Пусть синтаксис описания идентификаторов задается правилами:

$$DEC \to \textbf{real}\ IDS \mid \textbf{integer}\ IDS \mid \textbf{boolean}\ IDS$$
$$IDS \to id$$
$$IDS \to IDS, id,$$

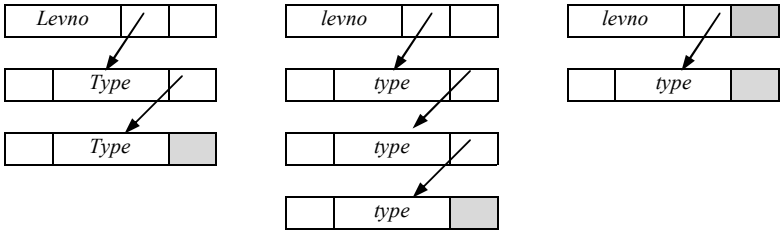
а блок определяется как

$$BLOCK \to \textbf{begin}\ DECS;\ STAT\ \textbf{end},$$

где

$$DECS \to DECS; DEC$$
$$DECS \to DEC$$
$$STATS \to STATS; st$$
$$STATS \to st$$

В этом случае структуру таблицы символов можно представить в виде



Таким образом, в любой точке разбора в цепи находятся те блоки, в которые делается текущее вхождение, а уже описанные идентификаторы помещаются в список идентификаторов для того блока, где они описаны.

Для описания таблиц задаются структуры строго фиксированной конфигурации. Обычно в этих структурах идентификаторы и типы представляются целыми числами. Имеется указатель на элемент таблицы символов, соответствующих наименьшему включающему блоку.

В языке, обладающем описанными выше четырьмя свойствами, в качестве структуры данных для таблицы символов удобно использовать стек, каждым элементом которого служит элемент этой таблицы символов.

При встрече с описанием соответствующий элемент таблицы символов помещается в верхнюю часть стека, а при выходе из блока все элементы таблицы символов, соответствующие описаниям в этом блоке, удаляются из стека. Указатель стека понижается до положения, которое он имел до вхождения в блок. В результате в любой момент разбора элементы таблицы символов, соответствующие всем текущим идентификаторам, находятся в стеке, а связанные с ними прикладные и

определяющие реализации идентификаторов требуют поиска в стеке в направлении сверху вниз.

Рассмотренный метод иллюстрируется следующим примером

Вид программы

```
begin int a, b
.
  begin int c, d
.
 end
  begin int e, f
 .
 end
end
```

f	**int**
e	**int**
b	**int**
a	**int**

Таким образом, включение действий в грамматику позволяет получить простой и элегантный компилятор. При этом действия выполняются на соответствующем уровне в грамматике.

Контрольные вопросы

1. Технология включения действий в грамматику.
2. Получение четверок, грамматика для четверок.
3. Разбор арифметического выражения с одновременной генерацией кода.
4. Метод определения контекстно – зависимых частей языка.
5. Синтаксис описания идентификаторов и блоков.
6. Структура таблицы символов.
7. Программная реализация таблицы символов.

8. Проектирование компиляторов

8.1. Число проходов

Разработчики компиляторов находят идею однопроходного компилятора привлекательной, так как не надо заботиться о связях между проходами, промежуточных языках и т.д. Кроме того, нет трудностей в ассоциировании ошибок программы с исходным тестом. Однако вопрос о количестве проходов неоднозначен. Прежде всего надо определить, что мы будем считать проходом.

Определение.

Если какая-либо фаза процесса компиляции требует полного прочтения текста, то это обычно называют проходом.

Проходы бывают прямыми или обратными, т.е. за один проход исходный текст можно считать слева направо или справа налево.

Большинство языков, использующих идею описания переменных до их первого использования (*Pascal, C++* и др.), либо использующих принцип умолчания, в принципе могут быть однопроходными. Однако есть ряд особенностей, которые не позволяют обеспечить компиляцию за один проход. Особенно ясно это можно продемонстрировать на проблеме компиляции взаимно рекурсивных процедур. Допустим, что тело процедуры *A* содержит вызов процедуры *B*, а процедура *B* содержит вызов процедуры *A*. Если процедура *A* объявляется первой, то компилятор не будет генерировать код для вызова *B* внутри *A*, не зная типов параметров *B*, и в случае процедуры, возвращающей результат, тип этого результата может потребоваться для идентификации обозначения операции. Единственное разумное решение данной проблемы – позволить компилятору сделать дополнительный проход перед генерацией кода.

Часто увеличение числа проходов компилятора используется искусственно - для уменьшения памяти, занимаемой компилятором в оперативном запоминающем устройстве (ОЗУ). Кроме того, количество проходов зависит не только от особенностей транслируемого языка, но и от используемой ЭВМ и операционного окружения.

Обычные традиционные трансляторы используют от *четырех* до *восьми* проходов, часть из которых тратится на оптимизацию загрузочного кода. Однако для рассмотрения принципов компиляции достаточно остановиться на одно – (максимум) двухпроходных компиляторах.

8.2. Таблицы символов

Информацию о типе (виде) идентификаторов синтаксический анализатор хранит с помощью таблицы символов. Этими таблицами также пользуются генератор кода для хранения адресов значений во время прогона. В языках, имеющих конечное число типов (видов), информацией о типе может быть простое целое число, представляющее этот тип, а в языках имеющих потенциально бесконечное число типов (*C++, Pascal* и др.), - указатель на таблицу видов, элементами которого являются структуры, представляющие вид.

Как уже отмечалось, для простых языков (*Fortran, Basic* и др.), имеющих конечное число типов, каждый из них может быть представ-

лен целым числом. Например, тип **integer** - посредством 1, а тип **real** – посредством 2. В этом случае таблица символов имеет вид массива с элементами

Identifier, type.

В языках, имеющих ограничение на число литер в идентификаторе, обычно в таблице символов хранятся сами идентификаторы, а не какое-либо их представление, полученное посредством лексического анализа.

С таблицей символов ассоциируются следующие действия.

1) Идентификатор, встречающийся впервые, помещается в таблицу символов, его тип определяется по соответствующему описанию.
2) Если встречается идентификатор, который уже помещен в таблицу, его тип определяется по соответствующей записи в таблице.

В соответствии с этой моделью идентификаторы будут появляться в таблице в том же порядке, в каком они впервые встречаются в программе. Всякий раз, когда анализатору встречается идентификатор, он проверяет, есть ли уже этот идентификатор в таблице, и при его отсутствии в конец таблицы вносится соответствующая запись. Если идентификатора в таблице нет, то требуется ее полный просмотр; но даже в тех случаях, когда идентификатор находится в таблице, поиск в среднем охватывает ее половину. Такой поиск в таблице обычно называют линейным. Естественно, что при больших размерах таблицы этот процесс может оказаться длительным.

Если бы идентификаторы были расположены в алфавитном порядке, то поиск осуществлялся бы гораздо быстрее за счет последовательного деления таблицы пополам (двоичный поиск), однако на сортировку записей по порядку каждый раз при добавлении новой записи уходило бы очень много времени, что существенно снижает достоинства этого метода. Поэтому при создании таблицы символов и поиска в ней обычно используется метод *хеширования.*

Для многих языков программирования в общем случае число возможных идентификаторов хотя и конечно, но очень велико.

В общем случае для таблицы символов используется массив из большего числа элементов, чем максимальное число ожидаемых идентификаторов в программе, и определяется отображение каждого возможного идентификатора на элемент массива (функция хеширования). Это отображение не является, конечно, отображением один к одному, а множество идентификаторов отображается на один и тот же элемент массива. Всякий раз при появлении идентификатора проверяется на-

личие записи в соответствующем элементе массива. При отсутствии записи этот идентификатор еще не находится в таблице символов, и можно сделать соответствующую запись. Если этот элемент не пустой, проверяется, соответствует ли запись в нем данному идентификатору, и когда выясняется, что соответствия нет, точно так же исследуется следующий элемент и т.д. до тех пор, пока не обнаружится пустой элемент или запись. Таким образом, таблица символов просматривается линейно, начиная с записи, полученной с помощью функции отображения, до тех пор, пока не встретится сам идентификатор или пустой элемент, указывающий на то, что для этого идентификатора записи не существует. Такой массив называется *таблицей хеширования*, функция отображения – *функцией хеширования*. Таблица хеширования обрабатывается циклично, т.е., если поиск не завершен к моменту достижения конца таблицы, он должен быть продолжен с ее начала.

Самая простая функция хеширования подразумевает использование первой буквы каждого идентификатора для его отображения на элемент 26-элементного массива. Идентификаторы, начинающиеся с *A,* отображаются на первый элемент массива, начинающиеся с *B* – на второй и т.д. После встречи с идентификаторами

CAR DOG CAB ASS EGG

таблица примет вид табл.7.1; позиции идентификаторов зависят от порядка их внесения в таблицу.

Если идентификатор не может быть внесен в ту позицию, которая задается функцией хеширования, происходит так называемый *конфликт*. Чем больше таблица (и меньше число идентификаторов, отражающихся на каждую запись), тем меньше вероятность конфликтов. При наличии в программе только вышеперечисленных идентификаторов и использовании рассмотренной функции хеширования таблица заполнялась бы неравномерно, и имела бы место *кластеризация*. Функция хеширования, которая зависела бы от последней литеры идентификатора, вероятно, меньше бы способствовала кластеризации. Конечно, чем сложнее функция, тем больше времени требуется для ее вычисления при внесении в таблицу идентификатора или при поиске конкретного идентификатора в таблице. Поэтому выбор функции хеширования – важная задача при построении компиляторов.

Таблица 7.1.

1	*ASS*	
2		
3	*CAR*	

4	*DOG*	
5	*CAB*	
6	*EGG*	
7		
.		

Обычно выход из конфликтов осуществляется циклической проверкой один за другим элементов таблицы до тех пор, пока не находится идентификатор или пустой элемент. Однако на практике используются и другие методы. Обычно вырабатывается некоторое правило, последовательное применение которого позволило бы (при необходимости) просмотреть все записи таблицы, прежде чем какая-либо из них встретится вторично. Действие, выполняемое функцией хеширования, называют *первичным хешированием*, а вычисление последующих адресов в таблице – *вторичным хешированием* или *перехешированием*. Функция перехеширования, рассматриваемая в примере, предполагает просто добавление единицы (циклично) к адресу таблицы. Эта функция, как и все функции хеширования, обладает следующим свойством: если n – некий адрес в таблице, а p – число элементов в таблице, то

$$n,\ rehash(n),\ rehash^2(n),\cdots rehash^{p-1}(n)$$

все являются адресами, и

$$rehash^p(n) = n\ .$$

Описанная выше функция перехеширования определяется процедурой

int procedure *rehash*(**int** *n*)
if *n*<*p* **then** *n*+1 **else** 1.

У этой функции есть тенденция создавать в таблице кластеры в той же мере, в какой вероятность кластеризации возрастает в связи с перехешированием. Весьма желательно, чтобы функция перехеширования могла находить адреса подальше от того, с которого начала. Если элементы таблицы простые числа (т.е. не имеют других делителей кроме 1), то вместо 1 функция перехеширования может добавлять к адресу любое положительное целое число h, такое что $h<p$; в результате бы имели

int procedure *rehash*(**int** *n*)
if *n*+*h*<*p* **then** *n*+*h* **else** *n*+*h*-*p*.

Подходящее значение h сводило бы кластеризацию к минимуму. Так как p - простое число, функция перехеширования выдаст последовательно все адреса в таблице, прежде чем она повторится.

Есть много других вариантов избежать перехеширования при создании таблиц идентификаторов. Рассмотрим некоторые из них.

- *Сцепление элементов.* В этом случае переполнения таблиц можно избежать путем использования указателей (рис. 7.1.)

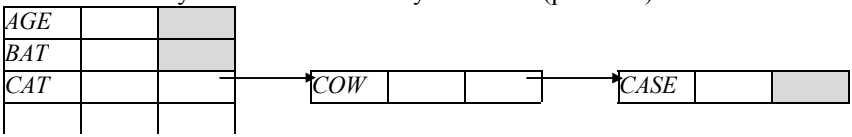


Рис. 7.1. Сцепление элементов

Для этого в таблице необходимо предусмотреть место для указателей, что ведет к увеличению объема последней.

- *Бинарное дерево*. Бинарное дерево (рис. 7.2.) состоит из некоторого количества вершин, каждая из которых содержит идентификатор, его тип и т.д., и двух указателей на другие вершины.

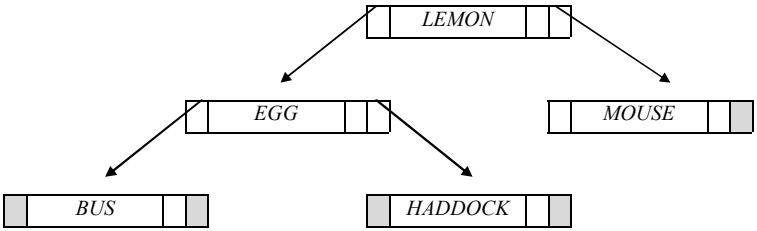


Рис. 7.2. Бинарное дерево

Бинарное дерево, приведенное на рис. 7.2, упорядочено в алфавитном порядке слева направо (т.е. его вершины расположены в алфавитном порядке их обхода изнутри). Поддерево любой вершины обозначается с помощью указателя. Поиск осуществляется с использованием рекурсивного алгоритма обхода: *пересечь левое поддерево, пройти корень, пересечь правое поддерево*. К бинарному дереву всегда можно добавить новую вершину, поместив ее в соответствующее место. Время поиска зависит от глубины дерева.

Расплачиваться за использование бинарного дерева в качестве таблицы символов приходится дополнительным объемом памяти, требуемым для указателей.

8.3. Таблица видов

В современных языках программирования число видов (абстрактных типов данных) потенциально бесконечно. Естественно, что в этом случае вид нельзя представить целым числом. В этой связи возникает проблема – найти приемлемый (с точки зрения разработчиков компилятора) способ представления любого возможного вида.

В существующих языках существует 5-7 вариантов видов:

1) *основные виды*, например **int, real, char, bool** и др.;
2) *длинные и короткие виды*, которые содержат символы **long** или **short**, появляющиеся перед основными видами;
3) *указатели* на адрес ячейки памяти, выделенной для данного вида;
4) *структурные виды*, типа **struct** и последовательностью полей; каждое поле имеет вид и селектор, обычно заключенные в скобки;
5) *виды массивов;*
6) *объединенные виды*, состоящие из символов **union** или **void**, используемых для выражения значений, которые могут принадлежать нескольким видам;
7) *виды процедур*, представленные символами **procedure, function** и др., используемых для выражения значений, являющихся процедурами.

Естественно было бы представить все виды каким-нибудь одним типом, например, структурой. Для представления вида можно использовать массив или список, причем список более удобен, так как компилятор обычно строит структуру вида слева направо при просмотре программы, и необходимое для представления каждого вида пространство неизвестно, когда встречается его первый символ.

Описатель

proc(**real, int**) **bool,**

выражающий значение вида «процедура-с-вещественными-и-целочисленными-параметрами-дающая-логический-результат» может быть представлена структурой с отдельными указателями на список параметров и результат (рис. 7.3).

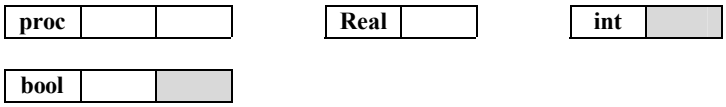


Рис. 7.3. Структура процедуры

Аналогичным образом вид

struct(**int**(*i*), **struct**(**int** *j*, **bool** *y*), **real** *r*)

может быть представлен так, как показано на рис. 7.4.

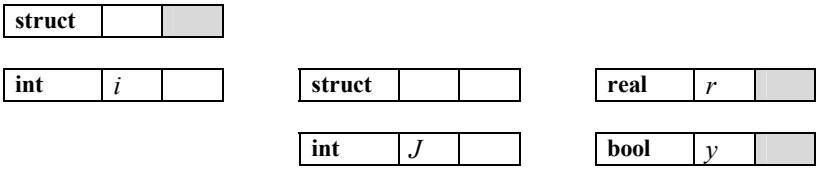


Рис 7.4. Структура типа **struct**

Каждая ячейка имеет два (возможно пустых) указателя; вертикальный указатель используется в случае структурных, объединенных и процедурных видов.

С помощью рассмотренного метода представления видов компилятор может легко выполнять следующие операции:

1) нахождение вида результата процедуры;
2) выбор структуры поля;
3) разыменование значения, т.е. замену адреса значением в адресе;
4) векторизацию, т.е. построение линейной структуры для любого массива.

Контрольные вопросы

1. Понятие прохода компилятора, необходимость использования нескольких проходов при компиляции.
2. Назначение таблицы символов.
3. Методы организации таблицы символов.
4. Хеширование, разрешение конфликтов при хешировании.
5. Сцепление элементов и бинарное дерево.
6. Назначение таблицы видов.
7. Способы организации таблицы видов.

9. Распределение памяти

9.1. Стек времени прогона

После выяснения структуры программы необходимо выделить место в памяти для внесения значений переменных и поместить соответствующие адреса в таблицу символов. Фаза распределения памяти практически не зависит от языка и машины и одинакова для большинства языков, имеющих блочную структуру. Распределение памяти за-

ключается в отображении значений, появляющихся в программе, на запоминающее устройство машины. Если реализуемый язык имеет блочную структуру, а ЭВМ имеет линейную память, то наиболее подходящим устройством, на котором будет базироваться распределение памяти, является стек или память магазинного типа.

Каждой программе для хранения значений переменных и промежуточный значений выражений необходим определенный объем памяти. Например, если идентификатор описывается как

int *x,*

т.е. *x* может принимать значение типа целого, то компилятору необходимо выделить память для *x*. Иными словами, компилятор должен выделить достаточно места, чтобы записать любое целое число. Аналогично, если *y* описывается как

struct (**int** *number*, **real** *size*, **bool** *n*) *y,*

то компилятор обеспечивает для значения *y* память с объемом, достаточным для хранения в нем целого, вещественного и логического значения. В обоих случаях компилятор не должен испытывать затруднений в вычислении требуемого объема памяти. Если *z* описывается

int *z*[10],

то объем памяти, необходимый для хранения всех элементов *z*, в 10 раз больше памяти для записи одного целого значения. Однако, если бы *w* был описан в виде

int *w*[*n*],

а значение *n* оказалось бы неизвестным во время компиляции (оно должно быть рассчитано программой), то компилятор не знал бы, какой объем памяти ему нужно выделить для *w*. Обычно *w* называют *динамическим* массивом. Память для *w* выделяется во время прогона. Память, выделяемую во время компиляции, называют *статической*, а во время прогона – *динамической*. В большинстве компиляторов память для массивов (даже имеющих ограничения констант) выделяется во время прогона, поэтому она считается динамической.

Память нужна также для промежуточных результатов и констант. Например, при вычислении выражения $a + c \times d$ сначала вычисляется $c \times d$, причем значение запоминается в машине, а затем выполняется сложение. Память, используемая для хранения результатов, называется *рабочей*. Рабочая память может быть статической и динамической.

В каждом компиляторе предусмотрена схема распределения памяти, которая до некоторой степени зависит от компилируемого языка. В Фортране память, выделяемая для значений идентификаторов, никогда не освобождается, так что подходящей структурой является одномер-

ный массив. Если считать, что массив имеет левую и правую стороны, память может выделяться слева направо. При этом применяется указатель, показывающий первый свободный элемент массива. Например, в результате описания

INTEGER *A, B, X, Y*

выделяется память, как это показано на рис. 8.1.

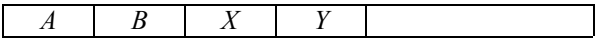


Рис. 8.1. Распределение памяти для Фортрана

Такая схема не учитывает тот факт, что рабочая память используется неоднократно и весьма неэффективна для языка с блочной структурой.

В языке, имеющем блочную структуру, память обычно высвобождается при выходе из блока, которому она выделена. В этом случае оптимальным решением было бы разрешить указателю отодвигаться влево при высвобождении памяти. Такой механизм распределения эквивалентен стеку времени прогона или памяти магазинного типа.

Пусть имеется программа вида:

```
begin real x, y
.
.
.
    begin int c, d
.
.
.
    end
    begin char p, q
.
.
.
     end
.
.
.
end
```

На рис. 8.2 показаны «моментальные снимки» стека времени прогона на различных этапах ее выполнения.

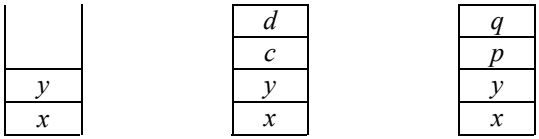


Рис. 8.2. Стек времени прогона

На рис. 8.2 изображено место, занимаемое значениями идентификаторов во время прогона. Часть стека, соответствующую определенному блоку, называют *рамкой* стека. *Указатель стека* показывает на его первый свободный элемент.

Кроме указателя стека требуется также указатель на дно текущей рамки (*указатель рамки*). При входе в блок этот указатель устанавливается равным текущему значению указателя стека. При выходе из блока сначала указатель стека устанавливается равным значению, соответствующему включающему блоку. Указатель рамки включающего блока может храниться в нижней части текущей рамки стека, образуя часть статической цепи или массива, который называется *дисплеем*. Его можно использовать для хранения во время прогона указателей на начало рамок стека, соответствующих всем текущим блокам (рис. 8.3).

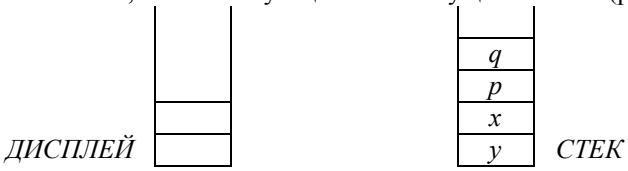


Рис. 8.3. Система «дисплей-стек»

Это упрощает настройку указателя при выходе из блока.

Если бы вся память была статической, адреса времени прогона могли бы распределяться во время компиляции, и значения элементов дисплея также были бы известны во время компиляции.

Рассмотрим пример программы:

begin int *n*; *read*(*n*);
    int *numbers*[*n*];
    real *p*;
    begin real *x, y*;

Место для *numbers* должно выделяться в первой рамке стека, а для *x* и *y* – в рамке над ней. Но во время компиляции неизвестно, где должна начинаться вторая рамка, так как неизвестен размер чисел. Одно из решений в этой ситуации – иметь два стека: один для статиче-

ской памяти, распределяемой в процессе компиляции, а другой для динамической памяти, распределяемой в процессе прогона.

Другое решение заключается в том, чтобы при компиляции выделять статическую память в каждом блоке в начале каждой рамки, а при прогоне – динамическую память над статической в каждой рамке. Это значит, что когда происходит компиляция, неизвестно, где начинаются рамки, но можно распределить статические адреса *относительно начала определенной рамки*. При прогоне точный размер рамок, соответствующих включающим блокам, известен, так что при входе в блок нужный элемент дисплея всегда можно установить так, чтобы он указывал на начало новой рамки (рис. 8.4).

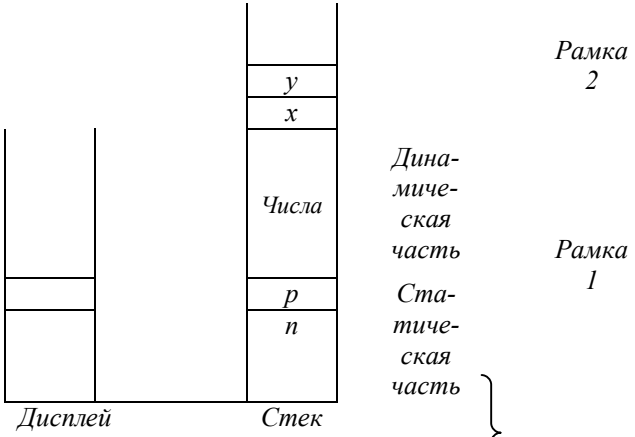


Рис. 8.4. Стек прогона для массива

В этой структуре массив занимает только динамическую память. Однако некоторая информация о массиве известна во время компиляции, например его размерность (а, следовательно, и число границ – две на каждую размерность), и при выборке определенного элемента массива она может потребоваться. В некоторых языках сами границы могут быть неизвестны при компиляции, но всегда известно их число, и для *значений* этих границ можно выделить статическую память. Тогда можно считать, что массив состоит из статической и динамической частей. Статическая часть массива может размещаться в статической части рамки, а динамическая – в динамической. Кроме информации о границах, в статической части может храниться указатель на сами элементы массива (рис. 8.5).

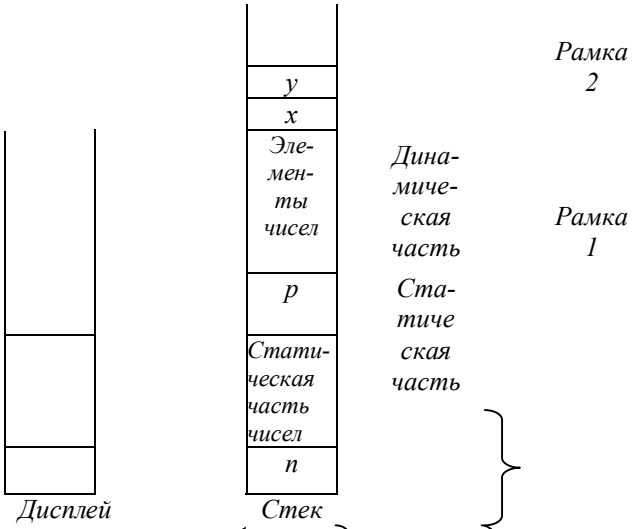


Рис. 8.5. Модифицированный стек прогона для массива

Когда в программе выбирается конкретный элемент массива, его адрес внутри динамической памяти должен вычисляться в процессе прогона. Пусть имеется массив

int *table*[1:10, -5:5].

Будем считать, что элементы массива записаны в лексикографическом порядке индексов, т.е. элементы таблицы хранятся в следующем порядке:

table[1, -5], *table*[1, -4]……. *table*[1, 5],
table[2, -5], *table*[2, -4]……. *table*[1, 5],
.
.
.
table[10, -5], *table*[10, -4]……. *table*[10, 5].

Адрес конкретного элемента вычисляется как смещение по отношению к базовому адресу (адресу первого элемента) массива:

$$ADDR(table[I,J]) = ADDR(table[l_1,l_2]) + (u_2 - l_2 + 1) \times (I - l_1) + (J - l_2).$$

Здесь l_1 и u_1 - нижняя и верхняя границы первой размерности и т.д. и считается, что элемент массива занимает единицу объема памяти.

Выражение $(u_i - l_i + 1)$ задает число различных значений, которые может принимать *i*-й индекс. Расстояние между элементами, отли-

чающимися на единицу в *i*-м индексе, называется *i-й шагом* и обозначается s_i. Пример шагов массива (рис. 8.6):

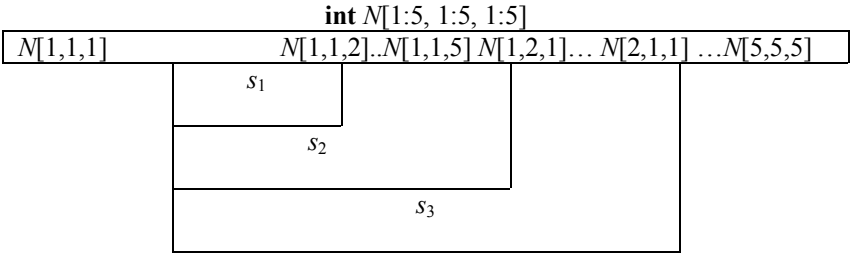


Рис. 8.6. Схема смещений

Если бы каждый элемент массива занимал объем памяти *r*, то эти шаги получили бы умножением всех вышеприведенных величин на *r*.

Ясно, что вычисление адресов элементов массива в процессе прогона может занимать много времени. Но шаги могут вычисляться только один раз и храниться в статической части массива наряду с границами. Такая статическая информация называется *описателем массива.*

Во многих языках все идентификаторы должны описываться в блоке, прежде чем можно будет вычислять какие-либо выражения. Отсюда следует, что рабочую память нужно выделять в конкретной рамке стека, над памятью, предусмотренной для значений, соответствующих идентификаторам (называемой *стеком идентификаторов*). Обычно статическая рабочая память выделяется в вершине статического стека идентификаторов, динамическая рабочая память – в вершине динамического стека идентификаторов.

В процессе компиляции статический стек идентификаторов растет по мере объявления идентификаторов. Вместе с тем, статический рабочий стек может не только увеличиваться в размерах, но и уменьшаться. Пример:

$$x=a+b\times c.$$

При вычислении выражения $(a+b\times c)$ потребуется рабочая память, чтобы записать $b\times c$ перед сложением. Ту же самую рабочую память можно использовать для хранения результатов сложения. Однако после осуществления операции присвоения этот объем памяти можно освободить, так как он уже не нужен.

Динамическая память должна распределяться во время прогона, статическая же распределяется во время компиляции. Объем статической рабочей памяти, который должен выделяться каждой рамке, оп-

ределяется не рабочей памятью, требуемой в конце блока, а *максимальной* рабочей памятью, требуемой любой точке внутри блока. Для статической рабочей памяти эту величину можно установить в процессе компиляции.

9.2. Методы вызова параметров

При распределении памяти в процессе компиляции и прогона необходимо организовать выделение памяти под переменные, объявленные в процедурах. Объем выделяемой памяти зависит от метода сообщения между фактическим параметром в вызове процедуры, и формальным параметром в описании процедуры. В различных языках используются различные методы «вызова параметров»; большинство языков предоставляет программисту возможность выбора по меньшей мере одного из двух методов.

Вызов по значению

Фактический параметр (которым может быть выражение) вычисляется, и копия его значения помещается в память, выделенную для формального параметра. В этом методе формальный параметр ведет себя как локальная переменная и принимает присвоение в теле процедуры. Такое присвоение не влияет на значение фактического параметра, поэтому данный метод нельзя применять для вывода результата из процедуры. Вызов по значению является эффективным методом передачи информации в процедуру, в которой используются большие массивы.

Вызов по имени

Этот метод заключается в текстуальной замене формального параметра в теле процедуры фактическим параметром перед выполнением тела процедуры. Там, где фактическим параметром является выражение, оно должно вычисляться всякий раз, когда в теле процедуры появляется соответствующий формальный параметр. Это – дорогой метод. С точки зрения реализации аналогичный результат можно получить с помощью специальной подпрограммы времени прогона для вычисления соответствующего фактического параметра. Вызов такой подпрограммы эффективно заменит каждое появление формального параметра в теле процедуры.

Вызов по результату

Как и в вызове по значению, при входе в процедуру выделяется память для значения формального параметра. Однако *никакое* начальное значение формальному параметру не присваивается. Тело процедуры может осуществлять присвоение значения формальному параметру, а при выходе из процедуры значение, которое в этот момент имеет формальный параметр, присваивается фактическому параметру.

Вызов по значению и результату

Этот метод представляет собой комбинацию вызова по значению и вызова по результату. Копирование происходит при входе в процедуру и при выходе из нее.

Вызов по ссылке

Здесь за адрес формального параметра принимается адрес фактического параметра, если последний не является выражением. В противном случае выражение вычисляется, и его значение помещается по адресу, выделенному для формального параметра. При таком методе получается тот же результат, что и при вызове по значению для выражений. Вызов по ссылке считается хорошим компромиссным решением и осуществлен во многих языках.

9.3. Обстановка выполнения процедур

При вызове процедуры необходимо вносить поправку в стек рабочего времени, чтобы рамка, соответствующая телу процедуры, была немедленно помещена над рамкой, соответствующей блоку, в котором содержится ее описание. Иначе адреса, введенные во время прогона (номер блока, смещение), будут относиться к другой рамке. На практике, чтобы не изменять стек при исключении из него нескольких рамок, можно изменить дисплей. Тогда доступ через него даст изменение стека. Это изменение должно выполняться сразу же после вычисления параметров (рис. 8.7).

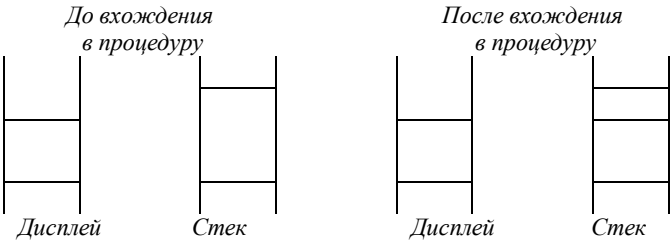


Рис. 8.7. Изменение дисплея при входе в процедуру

Конечно, после выхода из процедуры дисплей должен быть восстановлен.

Один из способов связи между фактическими и формальными параметрами заключается в введении дополнительного уровня (его называют *псевдоблоком*), в котором вычисляются фактические параметры. По завершению их вычисления рамку стека, соответствующую этому блоку, можно использовать в качестве рамки для тела процедуры после видоизменения дисплея. Это позволяет не прибегать к присвоению параметров.

Очевидно, что входы и выходы из блоков и процедур занимают много времени. Возникает вопрос, нельзя ли сократить время настройки дисплея, особенно для программ с множеством уровней блоков? Один из вариантов решения проблемы состоит в том, чтобы иметь единую рамку для всех значений стека. При этом полностью устраняются все издержки, связанные с выходом из блока и входом в него, но неперекрещивающиеся блоки не смогут пользоваться одной и той же памятью, и, следовательно, в обмен на сэкономленное время получается увеличение объема памяти.

Используя комбинацию рассмотренных вариантов, можно организовать работу компилятора в режиме оптимального времени либо оптимальной памяти.

9.4. «Куча»

В большинстве языков программирования обычная блочная структура обеспечивает высвобождение памяти в порядке, обратном тому, в котором она распределялась. Однако в программах со списками и другими структурами данных, содержащих указатели, часто необходимо сохранять память за пределами того блока, в котором она выделялась. Обычный список можно показать схематически, как на рис. 8.8.

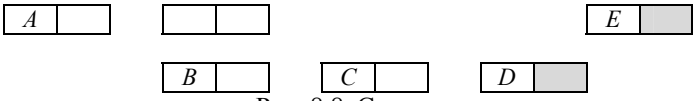


Рис. 8.8. Список

Каждый список состоит из головной части – литеры или указателя на другой список и хвостовой – указателя на другой список или нулевого списка. Список рис.8.8 можно записать в виде:

$$(A(BCD)E).$$

Скобки употребляются для разграничения списков и подсписков.

Память для любого элемента списка должна выделяться глобально, т.е. на время действия всей программы. Глобальная память не может ориентироваться на стек, поскольку его распределение и перераспределение не соответствует принципу «последним вошел – первым вышел». Обычно для глобальной памяти выделяется специальный участок памяти, называемый «кучей». Компилятор может выделять память и из стека, и из кучи, и в данном случае уместно сделать так, чтобы эти два участка «росли» навстречу друг другу с противоположных сторон запоминающего устройства (рис. 8.9).

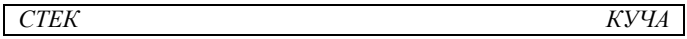


Рис. 8.9. Структура распределения памяти

Это значит, что память можно выделять из любого участка до тех пор, пока они не встретятся. Такой метод позволяет лучше использовать имеющийся объем памяти, чем при произвольном ее делении на два участка.

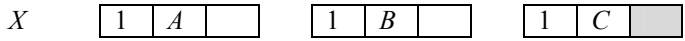
Размер стека увеличивается и уменьшается упорядоченно по мере входа в блоки и выхода из блоков. Размер же кучи может только увеличиваться, если не считать «дыр», которые могут появляться за счет освобождения отдельных участков памяти. Существует две разные концепции регулирования кучи. Одна из них основана на так называемых *счетчиках ссылок*, а другая – на *сборке мусора*.

9.5. Счетчик ссылок

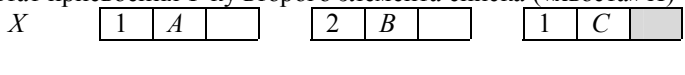
При использовании счетчика ссылок память восстанавливается сразу после того, как она оказывается недоступной для программы. Куча рассматривается как последовательность ячеек, в каждой из которых содержится невидимое для программиста поле (счетчик ссылок), в котором ведется счет числа других ячеек или значений в стеке, указы-

вающих на эту ячейку. Счетчики ссылок обновляются во время выполнения программы, и когда значение конкретного счетчика становится нулем, соответствующий объем памяти можно восстанавливать.

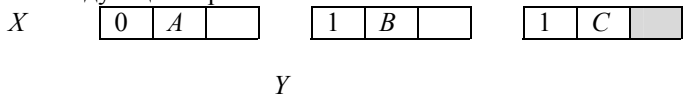
Для простоты допустим, что в каждой ячейке есть три поля, первое из которых отводится для счетчика ссылок. Если X – идентификатор, указывающий на список, то его значение может быть представлено



Результат присвоения Y-ку второго элемента списка («хвоста» X)



Результат следующего присвоения



На единицу уменьшился не только счетчик ссылок ячейки, на которую указывает X, но и ячейка, на которую указывает данная ячейка.

Алгоритм уменьшения счетчика ссылок после присвоения формулируется следующим образом: *уменьшить на единицу счетчик ссылок ячейки, на которую указывал идентификатор правой части присвоения; если счетчик ссылок является теперь нулем, следовать всем указателям этой ячейки, уменьшая счетчики ссылок до тех пор, пока (для каждого пути) не будет получено нулевое значение или достигнут конец пути.*

Поскольку нельзя следовать параллельно по двум или более путям, то потребуется стек для хранения в нем указателей, которым нужно еще следовать.

Счетчики ссылок в действительности нет необходимости хранить в самих ячейках. Их можно хранить где-нибудь в другом месте, лишь бы соблюдалось полное соответствие между адресом ячейки и его счетчиком ссылок.

Основными недостатками организации такого регулирования памяти являются:

1) Память, выделяемая для определенных структур, не восстанавливается с помощью описанного алгоритма, даже, если не бу-

дет доступа ни к одному из объемов памяти. Это четко видно на примере циклического списка (рис 8.10).

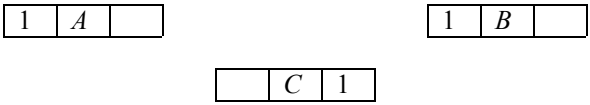


Рис. 8.10. Циклический список

Ни один из его счетчиков не является нулем, хотя никакие указатели на него извне не указывают. Этот объем памяти не восстановится никогда;

2) Обновление счетчиков ссылок представляет собой значительную нагрузку для всех программ. Это противоречит принципу Бауэра – «*простые программы не должны расплачиваться за дорогостоящие языковые средства, которыми не пользуются*».

9.6. Сборка мусора

Этот метод высвобождает память не тогда, когда она становится недоступной, а тогда, когда программе требуется память в виде кучи или в виде стека, но ее нет в наличии. Таким образом, у программ с умеренной потребностью в памяти необходимость в ее высвобождении может не возникнуть. Тем программам, которым не хватает объема памяти в виде кучи, придется приостановить свои действия и затребовать недоступный объем памяти, а затем продолжить свою работу.

Процесс, высвобождающий память, когда выполнение программы приостанавливается, называется *сборщиком мусора*. В него входят две фазы.

Фаза маркировки. Все адреса (или ячейки), к которым могут обращаться идентификаторы, имеющиеся в программе, маркируются путем изменения бита в либо самой ячейке, либо в отображении памяти в другом месте.

Фаза уплотнения. Все маркированные ячейки передвигаются в один конец кучи (в дальний от стека).

Фаза уплотнения не тривиальна, так как она может повлечь за собой изменение указателей. Однако в общем случае она достаточно алгоритмически проста.

Самым критическим фактором является объем рабочей памяти, имеющийся у сборщика мусора. Будет нелогично, если самому сборщику мусора потребуется большой объем памяти. Кроме того, жела-

тельно, чтобы сборщик мусора был эффективен по времени. Естественно, что оба эти параметра одновременно оптимизировать невозможно, поэтому необходимо выбирать компромисс.

Нахождение ячеек, доступных для программы, связано с прохождением по древовидным структурам, так как ячейки могут содержать указатели на другие структуры. Необходимо пройти по всем путям, представленным этими указателями, возможно, по очереди, а «очевидное» место хранения указателей, по которым еще придется пройти, будет находиться в стеке. Именно это и входит в функции алгоритма маркировки.

Для простоты будем считать, что каждая ячейка имеет максимум два указателя, т.е. память представляется массивом структур вида

struct (**int** *left, right,* **bool** *mark*)

Поля *left* и *right* – это целочисленные указатели на другие ячейки; для представления нулевого указателя используется нуль.

Алгоритм маркировки можно представить в виде процедуры *MARK*1, которая использует переменные:

STACK – стек, используемый сборщиком мусора, для хранения указателей;

T – указатель стека, указывающий на верхний элемент;

A – массив структур с индексами, начиная с 1, представляющий кучу.

После того как все отметки покажут значение «ложь», ячейки, на которые непосредственно ссылаются идентификаторы в программе, маркируются, и их адреса помещаются в *STACK*. Далее берется верхний элемент из *STACK*, маркируются непомеченные ячейки, на которые указывает ячейка, находящаяся по этому адресу, и их адреса помещаются в *STACK*. Выполнение алгоритма завершается, когда *STACK* оказывается пустым.

```
void proc MARK1;
begin int k;
      while T≠0
      do k= STACK[T];
         T minusab 1;
         if left of A[k]≠0 and not mark of A[left of A[k]]
            then mark of A[left of A[k]]=true;
            T plusab 1; STACK[T]=left of A[k]
         fi;
          if right of A[k]≠0 and not mark of A[right of A[k]]
            then mark of A[right of A[k]]=true;
            T plusab 1; STACK[T]=right of A[k]
```

```
        fi;
      od
end
```

До вызова этой процедуры куча могла иметь вид табл. 8.1, где маркировано только *A*[3], потому на него есть прямая ссылка идентификатора в программе. Результатом выполнения алгоритма будет маркировка *A*[5], *A*[1] и *A*[2] в указанном порядке.

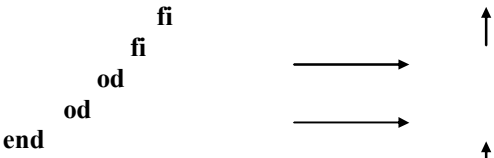
Таблица 8.1 – Состояние кучи

A	*левое*	*правое*	*маркировка*
5	2	1	**false**
4	1	2	**false**
3	5	1	**true**
2	3	0	**false**
1	5	0	**false**

Этот алгоритм очень быстрый, но крайне неудовлетворительный, во-первых, потому что может понадобиться большой объем памяти для стека, во-вторых, потому что объем требуемой памяти непредсказуем.

Другим крайним случаем является алгоритм, которому требуется небольшой фиксированный объем памяти, и не так важна эффективность по времени. Этот алгоритм представлен процедурой *MARK*2. Он нуждается в рабочей памяти для трех целых чисел: *k*, *k*1 и *k*2. Эта процедура просматривает всю кучу в поисках указателей от маркированных ячеек к немаркированным, маркирует последние и запоминает наименьший адрес ячейки, маркированной таким образом. Затем она повторяет этот процесс, начиная с наименьшего адреса, маркированного прошлый раз, и так до тех пор, пока при очередном просмотре не окажется ни одной новой маркированной ячейки.

```
void proc MARK2;
begin int k, k1, k2;
      k1=1;
      while k1≤M
      do k2= k1; k1=M+1;
         for k from k2 to M;
         do if mark then
               if left of A[k]=0 and not mark of A[left of A[k]];
                 then mark of A[left of A[k]]=true;
                      k1=min(left of A[k], k1)
               fi;
               if right of A[k]≠0 and not mark of A[right of A[k]]
                 then mark of A[right of A[k]]=true;
                      k1=min(right of A[k], k1)
```



Если этот алгоритм применяется в примере, который рассматривался для *MARK*1, то ячейки маркируются в том же порядке (5, 1, 2).

Оба эти алгоритма весьма неудовлетворительны (хотя и по разным причинам). Можно объединить их так, чтобы использовались преимущества каждого метода (алгоритм описан Кнутом).

Вместо стека произвольного размера, как в *MARK*1, применяется стек фиксированного размера. Чем он больше, тем меньше времени может занять выполнение алгоритма. При достаточно большом стеке его скорость сопоставима со скоростью *MARK*1. Однако, поскольку стек нельзя расширять, алгоритм должен уметь обращаться с переполнением, если оно произойдет. Когда стек заполняется, из него удаляется одно значение, чтобы освободить место для другого, добавляемого значения. Для запоминания удаленного таким образом из стека самого нижнего адреса используется целочисленная переменная (аналогично тому, что происходит в *MARK*2). Стек работает циклично с двумя указателями: один указывает вверх, другой вниз; это позволяет не перемещать все элементы стека при удалении из него одного элемента (рис. 8.11).

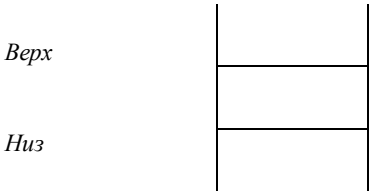


Рис 8.11. Стек с двумя указателями

Большей частью алгоритм работает как *MARK*1, и только когда стек становится пустым, завершает работу, если только из-за переполнения стека не были удалены какие-либо элементы. Если же элементы удалялись, то определяется самый нижний индекс, который «выпал» из стека, и именно с этого элемента начинается просмотр кучи. Этот процесс аналогичен процедуре *MARK*2. Любые другие адреса, которые нужно маркировать, помещаются в стек, и, если в конце просмотра он окажется пустым, алгоритм завершается. В противном случае алго-

ритм снова ведет себя как *MARK*1. Таким образом, алгоритм *MARK*3 действует аналогично процедуре *MARK*1, когда стек достаточно велик, и работает в смешанном режиме (*MARK*1 и *MARK*2) в противном случае.

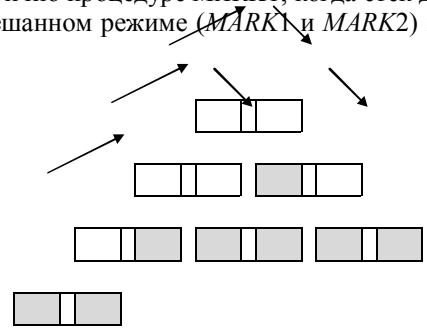


Рис. 8.12. Бинарное дерево

Еще один подход предложен Шорром и Уэйтом. Он отличается тем, что в фазе маркировки структуры, по которым придется проходить, временно изменяются, обеспечивая пути возврата, чтобы можно было обойти все пути. Благодаря этому можно не использовать стек произвольного размера. Допустим, необходимо пройти по бинарному дереву, показанному на (рис. 8.12). К тому моменту времени, когда фаза маркировки достигнет нижней левой ячейки, это дерево будет выглядеть так, как изображено на (рис. 8.13).

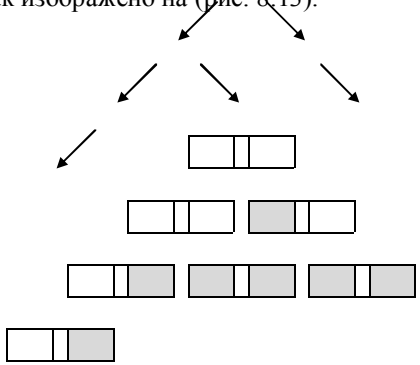


Рис. 8.13. Бинарное дерево в конце обхода

По завершению маркировки рассматриваемая структура примет свою первоначальную форму. Этот алгоритм требует одно дополнительное поле для каждой ячейки, в котором учитываются все пройденные пути. Как и в *MARK*1, время, затрачиваемое на выполнение алгоритма, пропорционально числу маркируемых ячеек.

Контрольные вопросы

1. Стек времени прогона, структура и программная реализация.
2. Технология распределения статической и динамической части стека.
3. Методы вызова параметров в процедурах, их достоинства и недостатки.
4. Организация распределения памяти при выполнении процедур.
5. Организация памяти для структур типа список (куча).
6. Освобождение памяти в «куче». Счетчик ссылок.
7. Сборка мусора. Алгоритмы *MARK*1 и *MARK*2. Их преимущества и недостатки.
8. Сборка мусора. Алгоритмы Кнута, Шорра и Уэйта.

10. Генерация кода

10.1. Генерация промежуточного кода.

Как отмечалось ранее, код генерируется при обходе дерева, построенного анализатором. Обычно в современных трансляторах генерация кода осуществляется параллельно с построением дерева, но может осуществляться и как отдельный проход. Генерация кода осуществляется в два этапа:

1) генерация не зависящего от машины промежуточного кода;
2) генерация машинного кода для конкретной ЭВМ.

Во многих компиляторах оба эти процесса осуществляются за один проход.

Обычно промежуточный код получается разбиением сложной структуры исходного языка на более удобные для обращения элементы.

Одним из распространенных видов промежуточного кода является четверки. Например, выражение

$$(-a+b)\times(c+d)$$

можно представить как четверки следующим образом:

$$-a=1$$
$$1+b=2$$
$$c+d=3$$
$$2\times 3=4.$$

Здесь целые числа соответствуют идентификаторам, присвоенным компилятором. Четверки можно считать промежуточным кодом высо-

кого уровня. Такой код называют трехадресным кодом (два адреса для операндов и один для результата).

Другой вариант кода – тройки (двухадресный код). Каждая тройка состоит из двух адресов операндов и знака операции.

Выражение

$$a + b + c \times d$$

можно представить в виде четверок:

$$a + b = 1$$
$$c \times d = 2$$
$$1 + 2 = 3$$

и виде троек:

$$a + b$$
$$c \times d$$
$$1 + 2 .$$

Если сам операнд является тройкой, то используется ее позиция (регистр) для хранения результата.

Как тройки, так и четверки можно распространить не только на выражения, но и на другие конструкции языка. Например, присвоение

$$a := b$$

в виде четверки представляется как

$$a := b = 1 ,$$

а в виде тройки – как

$$a := b .$$

Не менее популярны в качестве промежуточного кода префиксные и постфиксные нотации. В префиксной нотации каждый знак операции появляется перед своими операндами, а в постфиксной - после. Например, инфиксное выражение $a + b$ в префиксной нотации имеет вид $+ ab$, а в постфиксной - $ab +$.

Префиксная нотация известна также как польская запись, а постфиксная - как обратная польская запись (запись Лукашевича). Например, выражение

$$\left(a + b\right) \times \left(c + d\right)$$

в префиксной форме записывается следующим образом:

$$\times + ab + cd ,$$

а в постфиксной так:

$$ab + cd + \times .$$

В префиксной и постфиксной нотации скобки уже не употребляются, т.к. здесь никогда не возникает сомнения относительно того, какие

операнды принадлежат тем или иным знакам операций. В этих нотациях не существует приоритета знака операции.

Перегруппировку в результате преобразования $(a+b)\times(c+d)$ в $ab+cd+\times$ можно осуществить с помощью стека. При этом алгоритм сведется к трем действиям:

1) напечатать идентификатор, когда он встретится при чтении инфиксного выражения слева направо;
2) поместить в стек знак операции, когда он встретится;
3) когда встретится конец выражения (или подвыражения), выдать на печать этот знак операции, который находится в стеке.

Префиксные и постфиксные выражения можно также получать из представления выражения в виде бинарного дерева (рис. 9.1).

Пример: $(a+b)\times c+d$

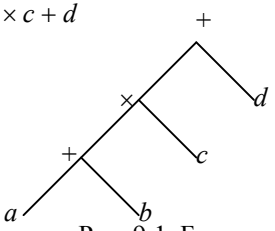


Рис. 9.1. Бинарное дерево

Чтобы получить представление префиксного выражения дерево обходят сверху в порядке:

- посещение корня;
- обход левого поддерева сверху;
- обход правого поддерева сверху,

что дает $+\times+abcd$.

Для получения постфиксного представления дерево обходят снизу:

- обход левого поддерева снизу;
- обход правого поддерева снизу;
- посещение корня.

В результате имеем: $ab+c\times d+$.

Далее все исследования будем проводить в терминах промежуточного языка

тип - команды параметры

Тип команды может быть, например, вызовом стандартного обозначения операции:

$$STANDOP\ \ II+, A, B, C\ .$$

Здесь *II+* - сложение двух целых чисел *A, B* и *C* служат во время прогона адресами двух операндов и результата. Для того чтобы в промежуточном коде можно было воспользоваться адресами во время прогона, распределение памяти к этому моменту должно быть уже закончено.

Промежуточный код напоминает префиксную нотацию в том смысле, что знак операции всегда предшествует своим операндам. Но он имеет менее общий характер, т.к. сами операнды не могут быть префиксными выражениями. При получении промежуточного кода для хранения адресов операндов до тех пор, пока не будет напечатан знак операции, используется стек. Поскольку знак операции можно установить лишь после того, как будут известны операнды, стек служит также для хранения каждого знака операции на то время, пока не определены оба операнда.

Адрес на время прогона соотносится со стеком, и каждый адрес можно представить тройкой вида

(тип – адреса, номер - блока, смещение).

Тип – адреса может быть прямым или косвенным (т.е. содержать значение или указывать на значение) и ссылаться на рабочий стек или стек идентификаторов.

Номер – блока позволяет найти номер уровня блока в таблице, что обеспечивает доступ к конкретной рамке блока через дисплей.

Смещение показывает смещение конкретной рамки по отношению к началу стека.

Адреса во время прогона для идентификаторов определяются в процессе распределения памяти и хранятся в таблице символов вместе с информацией о типе.

Кроме трехадресной команды существуют другие команды промежуточного кода:

*SETLABEL L*1

для установки метки и

ASSING type, *add*1, *add*2

для присваивания. Тип необходим, как параметр, чтобы определить размер значения передаваемого из *add*1 в *add*2.

10.2. Структура данных для генерации кода

Для хранения адресов операндов на время, пока их нельзя выдать, как параметры промежуточного кода, необходим стек значений. В

этом стеке, который называется *нижним стеком*, можно хранить и другую информацию, например:

- адрес времени прогона;
- тип данных;
- область действия (номер рамки).

Эта информация является *статической*, т.к. для большинства языков ее можно получить во время компиляции.

При трансляции *A+B* первыми помещаются в нижний стек статические свойства *A*. Любой элемент нижнего стека можно представить в виде структуры, имеющей поле для каждой из своих аналитических характеристик. Для идентификаторов аналитические характеристики находятся из таблицы символов. Затем в стек знаков операции помещается «+», и в нижний стек добавляются аналитические характеристики *B*. Знак операции берется из стека знаков операции, а его два операнда - из нижнего стека. Типы операндов используются для идентификации знака операции, после чего генерируется код. И, наконец, в нижний стек помещаются статические характеристики результата.

Этот процесс можно распространить на более сложные выражения, например, на грамматики с правилами.

$$
\begin{aligned}
EXP &\to TERM \mid \\
&\quad EXP + TERM \mid \\
&\quad EXP - TERM \\
TERM &\to FACT \mid \\
&\quad TERM \times FACT \mid \\
&\quad TERM / FACT \\
FACT &\to constant \mid \\
&\quad identifier \mid \\
&\quad (EXP)
\end{aligned}
$$

Для данных правил после чтения идентификатора или константы, знака операции и второго операнда необходимо выполнить следующие действия.

A1. После чтения идентификатора или константы (т.е. листа синтаксического дерева) поместить в нижний стек соответствующие характеристики.

A2. После чтения оператора поместить символ операции в стек знаков операций.

A3. После чтения правого операнда (который может быть выражением) извлечь из стеков знак операции и два его операнда, генерировать соответствующий код, т.к. знак операции

идентифицирован, и поместить в нижний стек статические характеристики результата. Тип результата становится известным во время идентификации знака операции (например, сложение двух целых чисел дает целое число).

При включении в грамматику этих действий она примет следующий вид:

$$
\begin{aligned}
EXP &\to TERM \mid \\
&\quad EXP + \langle A2 \rangle TERM \langle A3 \rangle \mid \\
&\quad EXP - \langle A2 \rangle TERM \langle A3 \rangle \\
TERM &\to FACT \mid \\
&\quad TERM \times \langle A2 \rangle FACT \langle A3 \rangle \mid \\
&\quad TERM / \langle A2 \rangle FACT \langle A3 \rangle \\
FACT &\to constant \langle A1 \rangle \mid \\
&\quad identifier \langle A1 \rangle \mid \\
&\quad (EXP)
\end{aligned}
$$

Нижний стек частично используется для передачи информации о типе по синтаксическому дереву.

Рассмотрим дерево для $a \times b + x \times y$ (рис. 9.2).

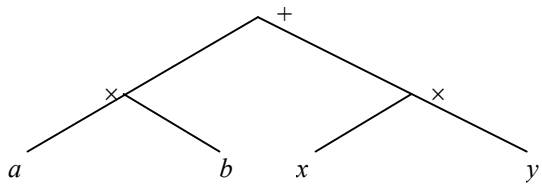


Рис. 9.2. Синтаксическое дерево для арифметического выражения

Если значения *a* и *b* имеют тип целого, а *x, y*-тип вещественного значения, компилятор может заключить, воспользовавшись информацией нижнего стека, что «+» вершины дерева представляет собой сложение целого и вещественного. Мы можем переписать выражение, расставив действия *A1, A2* и *A3* в том порядке, в каком они будут возникать при трансляции выражения.

a<A1>×<A2>b<A1><A3>+<A2>x<A1>×<A2>y<A1><A3><A3>.

Каждый вызов *A3* соответствует тому месту, где появился знак операции в постфиксной форме. Стек знаков операций служит для формирования постфиксной нотации. Поэтому последовательность

действий при трансляции данного выражения должна быть следующей.

A1. Поместить статические характеристики *a* в нижний стек.

A2. Поместить знак «×» в стек знаков.

A1. Поместить статические характеристики *b* в нижний стек.

A3. Извлечь статические характеристики *a* и *b* из нижнего стека и знак «×» из стека знаков операций, генерировать код для умножения двух целых чисел, поместить статические характеристики результата в нижний стек; тип результата - целый.

A2. Поместить знак «+» в стек знаков.

A1. Поместить статические характеристики *x* в нижний стек.

A2. Поместить знак «×» в стек знаков операций.

A1. Поместить статическую характеристику *y* в нижний стек.

A3. Извлечь статические характеристики *x* и *y* из нижнего стека и знак «×» из стека знаков операций, генерировать код умножения двух вещественных чисел, поместить статические характеристики результата в нижний стек; тип результата - вещественный.

A3. Извлечь два верхних элемента из нижнего стека и знак «+» из стека знаков операций, генерировать код сложения целого и вещественного значений, поместить статические характеристики результата в нижний стек; тип результата – вещественный.

Действия *A1, A2, A3* легко расширить, что позволит использовать большое число уровней приоритета для знаков операций и унарные знаки операций.

Нижний стек обеспечивает передачу информации вверх по синтаксическому дереву. Для передачи вниз по дереву используется *верхний стек*. Значение в него помещается всякий раз, когда во время генерации кода происходит вход в такую конструкцию, как присвоение или описание идентификатора. При выходе из этой конструкции значение из стека удаляется.

Еще одной структурой данных, которая требуется во время генерации кода, является таблица блоков (табл. 9.1.)

Таблица 9.1 – Таблица блоков

Блок	Уровень блока	Размер стека идентификаторов	Размер рабочего стека
1	1	14	16
2	2	12	11
3	2	21	13
4	3	4	9
5	2	6	12

В этой таблице есть запись для каждого блока программы, и эту запись можно рассматривать как структуру, имеющую поля, которые соответствуют номеру уровня блока, размеру статического стека идентификаторов, размеру статического рабочего стека и т.д. Такую таблицу можно заполнять во время прохода, генерирующего код, и с ее помощью во время следующего прохода вычислять смещения адресов рабочего стека по отношению к текущей рамке стека.

Таким образом, во время генерации кода используются следующие структуры данных:

- нижний стек;
- верхний стек;
- стек значений операций;
- таблица блоков;
- таблица видов и таблица символов из предыдущего прохода.

10.3. Генерация кода для типичных конструкций

10.3.1. Присвоение

destination := source

Значение, соответствующее *источнику*, присваивается значению, которое является адресом:

$p := x + y$ (значение $x + y$ присвоить p)

Допустим, что статические характеристики источника и получателя находятся в вершине нижнего стека. Последовательность действий: из нижнего стека удаляются два верхних элемента, затем выполняются следующие операции.

Проверяется непротиворечивость типов получателя и источника. Так как получатель представляет собой адрес, источник должен давать что-нибудь приемлемое для присвоения этому адресу. В зависимости от реализуемого языка типы *получателя* и *источника* можно определенным образом менять до выполнения операции присвоения. (Если источник — целое число, то его можно сначала преобразовать в вещественное, а затем присвоить адресу, имеющему тип вещественного числа.)

Там, где необходимо, проверяются правила области действия (для некоторых языков *источник* не может иметь меньшую область действия, чем *получатель*).

```
begin pointer real xx
begin real x
xx := x
end
```

присвоение недопустимо, и это может быть обнаружено во время компиляции, если в таблице символов или в нижнем стеке имеется информация об области действия.

Генерируется код присвоения, имеющий форму

ASSIGN type, S, D,

где *S* — адрес *источника*, *D* — адрес получателя.

Если язык ориентирован на выражения (то есть само присвоение имеет значение), статические характеристики этого значения помещаются в нижний стек.

10.3.2. Условные зависимости

if *B* ***then*** *C* ***else*** *D*

При генерации кода для такой условной зависимости во время компиляции выполняются три действия. Грамматика с включенными действиями имеет вид:

CONDITIONAL → ***if*** *B<A1>* ***then*** *C<A2>* ***else*** *D<A3>*

Действия *A1, A2, A3* (next — значение номера следующей метки, присваиваемое компилятором) означают:

A1. Проверить тип *B*, применяя любые необходимые преобразования типа для получения логического значения. Выдать код для перехода к *L<next>*, если *B* есть «ложь»:

JUMPF L<next>, <address of B>

Поместить в стек значение next (обычно для этого используется стек знаков операций). Увеличить *next* на 1. (Угловые скобки используются для обозначения значений величин, заключенных в них).

A2. Генерировать код для перехода через ветвь ***else*** (то есть переход к концу условной зависимости):

GO TO L<next>.

Удалить из стека номер метки (помещенный в стек действием *A1*), назвать ее *i*, генерировать код для размещения метки:

SET LABEL L<i>.

Поместить в стек значение *next*. Увеличить *next* на 1.

A3. Удалить из стека номер метки (*j*). Генерировать код для размещения метки:

SET LABEL L<j>.

Если условная зависимость сама является выражением, компилятор должен знать, где хранить его значение, независимо от того, какая часть является ***then*** или ***else***.

Аналогично можно обращаться с вложенными условными выражениями.

10.3.3. Описание идентификаторов

Допустим, что типы всех идентификаторов выяснены в предыдущем проходе и помещены в таблицу символов. Адреса распределяются во время прохода, генерирующего код. Перечислим действия, выполняемые во время компиляции.

В таблице символов производится поиск записи, соответствующей идентификатору *x*.

Текущее значение указателя стека идентификаторов дает адрес, который нужно выделить под x. Этот адрес

(*idstack, current block number, instack pointer*)

включается в таблицу символов, а указатель стека идентификаторов увеличивается на статический размер значения, соответствующего *x*.

Если *x* имеет динамическую часть (например, массив), то генерируется код для размещения динамической памяти во время прогона.

10.3.4. Циклы

Рассмотрим простейший пример цикла:

for *i* ***to*** 10 ***do*** *something.*

Для генерации кода требуется четыре действия, которые размещаются следующим образом:

for *i <A1>* ***to*** 10 *<A2>* ***do*** *<A3> something <A4>.*

Эти действия таковы.

A1. Выделить память для управляющей переменной *i*. Поместить сначала в эту память 1:

*MOVE «*1*», address* (управляющая переменная).

A2. Генерировать код для записи в память значения верхнего предела рабочего стека:

MOVE address (ulimit) (wostack, current block number, wostack pointer)

(*wostack pointer* — указатель рабочего стека).Увеличить указатель рабочего стека и уменьшить указатель нижнего стека, где хранились статические характеристики верхнего предела.

A3. Поместить метку

SET LABEL L<next>.

Увеличить *next* на 1.

Выдать код для сравнения управляющей переменной с верхним пределом и перейти к *L<next>*, если управляющая переменная больше верхнего предела:

JUMPG L<next>, address (controlled variable), address (ulimit).

Поместить в стек значение *next*. Поместить в стек значение (*next* – 1). Увеличить *next* на 1.

A4. Генерировать код для увеличения управляющей переменной

PLUS address (controlled variable), 1.

Удалить из стека номер (*i*). Генерировать код для перехода к *L*(*i*):

GO TO L<i>.

Удалить из стека номер метки (*j*). Поместить метку в конец цикла:

SET LABEL L<j>.

Таким образом, цикл

for *i* ***to*** 10 ***do*** *something*

генерирует код следующего вида:

MOVE «1», *address (controlled variable)*
MOVE address (ulimit), wostack pointer
*SET LABEL L*1
*JUMPG L*2 *address (controlled variable), address (ulimit)*
(something)
*GO TO L*1
*SET LABEL L*2.

Действия *A4* можно видоизменять, если приращение управляющей переменной будет не стандартным (1), а иным.

for *i* ***by*** *5* ***to*** *10* ***do.***

Для этого придется вычислять приращение и хранить его значение в рабочем стеке, чтобы использовать как приращение.

Если цикл содержит часть ***while,*** то

for *i* ***to*** *10* ***while*** *a<b* ***do.***

Действие *A3* следует видоизменить, чтобы при принятии решения о выходе учитывалось как значение части ***while***, так и управляющей переменной, причем любая из этих проверок достаточна для завершения цикла.

10.3.5. Вход и выход из блока

При входе в блок предположим, что во время предыдущего прогона получены таблицы символов и видов, дающие типы и виды всех идентификаторов. Тогда при входе в блок необходимо выполнить следующие основные действия.

Прочитать в таблице символов информацию, касающуюся блока, и связать ее с информацией включающих блоков таким образом, чтобы можно было выполнять «внешние» поиски определяющих реализаций идентификаторов.

Поместить в стек (*idstack pointer*). Поместить в стек (*wostack pointer*). Поместить в стек (*block number*). Все эти значения ссылаются на включающий блок и могут потребоваться вновь после того, как будет покинут блок, в который только что осуществлен вход:

Idstack pointer := 0

wostack pointer:= 0.

Генерировать код для направления *DISPLAY*.

BLOCK ENTRY block number.

Увеличить номер уровня блока на 1. Увеличить *ghn* (наибольший использованный до сих пор номер блока) на 1 и присвоить это значение номеру блока.

Прочитать информацию о видах и добавить ее в таблицу видов (если язык использует сложные виды (структуры)).

При выходе из блока из блока необходимо выполнить следующие действия.

Обновить таблицу блоков, задав размер стека идентификаторов и т.п.. для покинутого блока.

Исключить информацию в таблице символов для покинутого блока.

Генерировать код для изменения дисплея:

BLOCK EXIT block number.

Удалить из стека (*block number*). Удалить из стека (*wostack pointer*). Удалить из стека (*idstack pointer*). Уменьшить уровень блока на 1.

Поместить результат (при необходимости) в рамку стека вызывающего блока.

10.3.6. Прикладные реализации

Во время компиляции в соответствии с прикладной реализацией идентификатора, например *x* в *x* + 4, необходимо:

1) найти в таблице символов запись, соответствующую определяющей реализации (**int** *x*) идентификатора;
2) поместить в нижний стек статические характеристики, соответствующие идентификатору.

Подразумевается, что в нижний стек также помещаются статические характеристики констант и т.д.

10.4. Проблемы, связанные с типами

Основной проблемой для трансляторов с языков высокого уровня является приведение (автоматическое изменение) типов. Здесь можно выделить, как минимум, шесть задач.

1) Распроцедуривание- переход от **procedure real** к **real.**
2) Разыменование, например переход от **pointer real** к **real.**
3) Объединение, например переход от **real** к **strutct(real,char**).
4) Векторизация, например переход от **real** к **real [ ]**.
5) Обобщение, например переход от **int** к **real.**
6) Чистка, например переход от **real** к **void.**

Возможность осуществления приведения зависит от синтаксической позиции. Например, в левой части присвоения может иметь место только распроцедуривание (вызов процедур без параметров), а в правой части - любое из шести приведений. Иногда возникает необходимость нескольких приведений. Например, если *x* имеет вид **pointer real** и *a* –**pointer int**, то прежде чем производить присвоение *x*=*a*, необходимо сначала разыменовать, а затем обобщить.

В зависимости от того, какие приведения могут выполняться в синтаксических позициях, последние называются мягкими, слабыми, раскрытыми, крепкими и сильными. Например, левая часть присвоения называется мягкой (допускает только распроцедуривание), а правая часть - сильной (допускает любое приведение). Кроме ограничений типов приведений, разрешаемых в заданной синтаксической позиции, существуют правила, определяющие порядок осуществления различных приведений. Например, объединение может произойти только один раз и не должно следовать за векторизацией. Можно определить грамматику, которая генерирует все допустимые последовательности приведений в заданной синтаксической позиции, например:

SOFT => deprocedure |

deprocedure SOFT

Любое предложение, генерированное посредством *SOFT*, представляет собой допустимую последовательность приведений в мягкой синтаксической позиции (т.е. в левой части присвоения).

Для раскрытия позиции (например, индекса в *a*[*i*]) справедливы следующие правила:

MEEK => deprocedure |
deprocedure MEEK |
dereference |
dereference MEEK.

Другими словами, в раскрытой позиции можно выполнять распроцедуривание и разыменование любое число раз и в любом порядке, например:

pointer procedure poiter int в вид **int**

Для сильной позиции (например, правая часть присвоения или параметр в вызове процедуры) правила таковы:

STRONG => dereference STRONG |
deprocedure STRONG |
unit |
unit ROW |
widen |
widen widen |
widen ROW |
widen widen ROW |
ROW |
ROW => row |
row ROW

Вид данных до выполнения приведений называется априорным, а после выполнения - апостериорным. В случае сильных и раскрытых синтаксических позиций известны и априорный и апостериорный виды. Для других позиций известен лишь априорный вид и некоторая информация об апостериорном виде, например о том, что он должен начинаться со **struct** или **pointer struct**, или о том, что он не должен начинаться с **proc,** как в левой части присвоения.

Компилятор, применяя соответствующую грамматику, генерирует последовательность приведений из априорного вида к известному либо к подходящему апостериорному виду. Если нельзя найти никакой последовательности приведений, программа синтаксически неправильная.

С другой стороны, если подходящая последовательность существует, компилятор, применяя приведения по порядку, генерирует код времени прогона.

Еще один вид приведения - чистка. Чистка представляет собой особую форму приведения и происходит в тех местах, где стоит точка с запятой

x=y;

Еще одна сложность связана с выбирающим предложением. В предложении

x + ***if*** *b* ***then*** *1* ***else*** *2.3*

во время компиляции необходимо знать тип (вид) правого операнда знака «+». Все варианты выбирающего предложения должны приводить к общему виду, называемому объектным. Этот процесс называется уравнением, и его правила подразумевают, что последовательность сильных приведений можно применять во всех вариантах, кроме одного, в котором используется лишь последовательность приведений, уместных лишь для синтаксической позиции выбирающего предложения. В вышеприведенном примере выбирающее предложение находиться в крепкой синтаксической позиции, которая не допускает расширения. Однако внутри выбирающего предложения один вариант допускает сильное приведение, что может повлечь за собой расширение. В этом случае объектным видом окажется **real**, и первый вариант следует расширить, а второй нежелательно подвергать приведению.

Действия компилятора при обращении с выбирающими предложениями заключаются в том, что статические характеристики всех вариантов выбирающего предложения помещаются в нижний стек, а затем выводятся объектный вид и различные последовательности приведения для каждого варианта. Если какая-либо последовательность вызывает необходимость генерации кода во время прогона, ее можно выделить в отдельный поток, и между двумя этими потоками ввести указатели, чтобы во время следующего прохода код можно было соединить в нужном порядке.

10.5. Время компиляции и время прогона

Как отмечалось ранее, генератор кода во время компиляции обращается к нижнему стеку и генерирует код операций, которые будут выполняться во время прогона. Что именно должно быть сделано во

время компиляции, а что во время прогона существенно зависит от языка программирования. Тем не менее, разработчик компилятора имеет возможность разделять функции компиляции и прогона. Например, не вполне ясно, повлечет ли разыменование за собой какие-либо действия при прогоне или нет. На первый взгляд может показаться, что нет, так как это просто замена в памяти одного значения другим и изменение адреса времени прогона. Новым адресом будет тот, на который указывает первоначальный адрес. Однако, значение указателя может быть неизвестным при компиляции, новый адрес можно обозначить, изменив первоначальный адрес так, чтобы в нем указывался дополнительный косвенный уровень. Т.е. в некоторых случаях требуется выполнить действие (переслать значение по новому адресу).

Для повышения эффективности выдаваемого кода при компиляции можно проделать дополнительную работу, которую показывают компиляцией. В оптимизацию входит удаление кодов из циклов там, где это не влияет на значение программы, исключение возможности вычисления идентичных выражений более одного раза и т.д. Особое внимание уделяется возможности избежать перезаписи сложных структур данных во время прогона.

Контрольные вопросы

1. Технология создания промежуточного кода. Виды промежуточного кода.
2. Алгоритм преобразования арифметического выражения в префиксную и постфиксную форму.
3. Формализация записи промежуточного кода.
4. Структуры данных для генерации промежуточного кода.
5. Алгоритм генерации промежуточного кода для арифметических выражений.
6. Таблица блоков, ее назначение.
7. Генерация кода для присвоения.
8. Генерация кода для условных зависимостей.
9. Генерация кода для описания идентификаторов.
10. Генерация кода для циклов.
11. Генерация кода для входа и выхода из блока.
12. Генерация кода для прикладной реализации.
13. Проблемы генерации кода, связанные с типами.
14. Работа генератора кода во время компиляции и во время прогона.

11. Исправление и диагностика ошибок

11.1. Типы ошибок

Если программа, представленная компилятору, написана с ошибками, не на «исходном» языке, «недружелюбный» компилятор может просто проинформировать пользователя об этом, не указав, где произошла ошибка. Большинство пользователей не удовлетворит такой подход, поскольку они ожидают от компилятора:

1) точного указания, где находится (первая) ошибка программирования;
2) продолжения компиляции (или, по крайней мере, анализа) программы после обнаружения первой ошибки с целью обнаружения остальных.

Основные причины возникновения ошибок программирования можно классифицировать следующим образом.

Программист не совсем понимает язык, на котором он пишет, и использует неправильную конструкцию программы.

Программист недостаточно осторожен в применении конструкции языка и забывает описать идентификатор или согласовать открывающую скобку с закрывающей и т.д.

Программист неправильно пишет слово языка или какого-либо другой символ в программе.

Ошибки, обусловленные этими тремя факторами, по-разному обнаруживаются компилятором. Ошибки первого типа вылавливаются синтаксическим анализатором, и генерируется сообщение с указанием того символа, на котором поток программы стал недействительным. Ошибки второго типа распадаются на две категории. Те, которые относятся к контекстным свойствам языка (отсутствие идентификатора и д.р.), обнаруживаются процедурой выборки из таблицы символов во время синтаксического анализа. Такие ошибки, как недостающие скобки, обнаруживаются самим анализатором во время выполнения фазы одного из проходов. Ошибки третьего типа обычно выявляются во время лексического анализа.

Существуют ошибки еще одного типа, когда программа пытается выполнить деление на ноль или считывание за пределами файла. Они называются ошибками времени прогона, и обычно их нельзя обнаружить в процессе компиляции. Наша задача проанализировать методы диагностики всех видов ошибок фазы компиляции и рассмотреть методы их коррекции.

11.2. Лексические ошибки

Задача лексического анализатора – сгруппировать последовательность литер в символы исходного языка. При этом он работает исключительно с локальной информацией. В его распоряжении имеется небольшой объем памяти, и он не осуществляет предварительного просмотра. В тех случаях, когда лексический анализатор окажется не в состоянии сгруппировать какие-либо последовательности литер в символы (лексемы), будут возникать ошибки. Лексические ошибки можно разделить на следующие группы.

Одна из литер оказывается недействительной, т.е. она не может быть включена ни в один из символов. В таком случае лексический анализатор либо игнорирует эту литеру, либо заменяет ее какой-либо другой.

При попытке собрать выделенное слово языка выясняется, что последовательность букв не соответствует ни одному из этих слов. В этом случае можно воспользоваться алгоритмом подбора слова, чтобы идентифицировать слово, имеющее несколько другое написание. Например, *realab* представить как **real** *ab*.

Собирая числа, лексический анализатор может испытывать затруднения с последовательностью вида 42.34.41. Возможное решение здесь – допустить, какая бы ошибка не была, что предполагалось одно число, и предупредить программиста, что вместо этого числа принято конкретное число по умолчанию.

Отсутствие в программе какой-либо литеры приводит к тому, что лексический анализатор не может отделить один символ от другого. Например, если в *A+B* пропущен знак «+», то лексический анализатор просто пропустит идентификатор *AB*, не оповещая об ошибке на этой стадии. Однако отсутствие знака «+» в 1+*A* вызовет ошибку, хотя лексический анализатор не будет знать, к какой группе ошибок отнести 1*A* – к недопустимым идентификаторам или еще чему-либо.

Обычно проблему для лексического анализатора создают недостающие кавычки строки символов

string *food=”BREFD*

Следовательно, в остальной части программы открывающие и закрывающие кавычки могут быть перепутаны. В результате обрушится лавина сообщений об ошибках. «Смышленый» анализатор смог бы обнаружить неправдоподобную последовательность литер внутри кавычек (например, **end**) и исправить ошибку, поставив в нужном месте кавычки.

Многие компиляторы завершают свою работу тем проходом, где обнаружена ошибка. Однако современная тенденция построения компиляторов базируется на принципе обнаружения максимального числа ошибок. Таким образом желательно, чтобы компилятор продвинулся в своей работе как можно дальше. Поэтому лексический анализатор должен передать следующему проходу (фазе) последовательность действительных символов (а необязательно действительную последовательность символов). Для правильных в лексическом смысле программ это не представляет трудностей. При лексически неправильных программах приходится или игнорировать последовательность символов, или включать дополнительные. Может потребоваться изменить написание символов, разбить строки на действительные символы. Игнорировать последовательность литер - самое простое средство, но оно практически всегда приводит к возникновению синтаксических ошибок. Методы исправления лексическим анализатором недопустимых входов зависят от обстоятельств, и на практике их выбор определяется компилируемым языком.

11.3. Ошибки в употреблении скобок

Ошибки, связанные с употреблением скобок, обнаруживаются относительно легко. Обычно компиляторы содержат фазу, предшествующую полному синтаксическому анализу, на которой производится согласование скобок. Если применять скобки только одного типа, например «(» и «)», проверку можно осуществлять с помощью целочисленного счетчика. Этот счетчик первоначально устанавливается на нуль, затем увеличивается на единицу для каждой открывающей скобки и уменьшается на единицу для каждой закрывающей скобки. Последовательность скобок считается допустимой в том случае, когда:

1) счетчик ни при каких обстоятельствах не становится отрицательным;
2) при завершении работы счетчик будет на нуле.

В большинстве языков программирования встречаются различные типы скобок, например

{	}
[	]
begin	**end**
if	**fi**
case	**esac**

В этом случае необходимо согласовывать каждую закрывающую с соответствующей открывающей скобкой. Алгоритм согласования ско-

бок читает скобочную структуру слева направо, помещая каждую открывающую скобку в вершину стека. Когда встречается закрывающая скобка, соответствующая открывающая скобка удаляется из стека. Последовательность скобок считается допустимой, если:

1) при чтении закрывающей скобки не окажется, что она не соответствует открывающей, помещенной в вершине стека;
2) при завершении работы стек станет пустым.

Ошибка в употреблении скобок должна отразиться в четком сообщении, типа

BRACKET MISMATCH.

Если ошибка возникла из-за того, что не достает закрывающей скобки, то тип последней можно вывести на основании той скобки, которая находится в вершине стека. Один из возможных путей исправления ошибки заключается в том, что берется предполагаемая недостающая закрывающая скобка, открывающая скобка удаляется из стека, и выдается сообщение с указанием предполагаемого источника ошибки.

Диагностическое сообщение появится, однако, не в том месте, где был допущен пропуск скобки, так как ошибка останется незамеченной до тех пор, пока не встретится другая закрывающая скобка иного типа. При продолжении синтаксического анализа желательно, чтобы скобочная структура была исправлена. Пример

if *b* **then** *x* **else** (*p*+*q*×*r***2 **fi.**

Здесь пропущена закрывающая скобка «)». Это не обнаружится до тех пор, пока не встретиться **fi.** Однако неясно, где должна стоять эта скобка: после *r*, после *q*, после *p* или 2. Выяснить, что предполагал программист, невозможно. Поэтому самый легкий способ «исправления» - поставить закрывающую скобку непосредственно перед **fi.** Синтаксический анализатор продолжает работать, а на выход выдается сообщение о введенных изменениях.

11.4. Синтаксические ошибки

Термин «синтаксическая ошибка» употребляется для обозначения ошибки, обнаруживаемой контекстно-свободным синтаксическим анализатором. Современные анализаторы обладают этим важным свойством – обнаруживать синтаксически неправильную программу на первом недопустимом символе, т.е. они могут генерировать сообщения при чтении символа, который не должен следовать за прочитанной к тому времени последовательностью символов.

Ошибка в виде пропуска или неправильного употребления, допущенная на более раннем этапе, может проявиться совсем в другом месте программы. Это можно проиллюстрировать следующим примером.

while *x*>*y* ; **begin** *something* **end.**

Никакого сообщения об ошибке при встрече «;» на данной стадии синтаксический анализатор не выдаст. Последствия ее могут появиться на более поздней фазе анализа.

Сообщив о синтаксической ошибке, анализатор в большинстве случаев постарается продолжить разбор. Для этого ему понадобится исключить какие-либо символы, включить какие-либо символы или изменить их. Существует ряд стратегий исправления ошибок. Практически все они хорошо срабатывают в одних случаях и плохо – в других. «Хорошая» стратегия заключается в том, чтобы обнаружить как можно больше синтаксических ошибок и генерировать как можно меньше сообщений в связи с каждой синтаксической ошибкой. Обычно наилучшими методами являются методы, зависимые от языка, т.е. от знания исходного языка и от того, как он употребляется.

11.5. Методы исправления синтаксических ошибок

Режим переполоха

Один их наиболее распространенных методов исправления синтаксических ошибок носит название *режим переполоха*. При появлении недопустимого символа весь последующий исходный текст, вплоть до соответствующего ограничителя (например «;» или **end**), игнорируется. Ограничитель заканчивает какую-то конструкцию языка, и элементы удаляются из стека разбора до тех пор, пока не встретится адрес возврата. Этот элемент тоже удаляется из стека, а разбор продолжается, начиная с адреса в таблице разбора, содержащего следующий входной символ. Такой метод довольно легко реализуется, но имеет серьезный недостаток: длинные последовательности кода, соответствующие игнорируемым символам, не анализируются.

Исключение символов

Этот метод также легко реализуется и не требует изменения степени разбора. Когда считывается недопустимый символ, и он сам, и все последующие символы исключаются из исходной строки до тех пор, пока не встретится допустимый символ. Хотя при таком методе могут исключаться длинные последовательности, в отдельных случаях он весьма эффективен. Например, в

c := *d*+3; **end**,

где «;» является недопустимой, исправление ошибки – идеальное. Однако исключение скобок обычно разрушает блочную структуру и приводит к дальнейшим синтаксическим ошибкам.

Включение символов

Некоторые синтаксические анализаторы имеют наготове множество действительных символов продолжения. В некоторых случаях оправдано исправление программ путем подстановки одного из таких символов перед недопустимым символом, который вызвал ошибку. Например, последовательность

end begin

никогда не будет допустимой. Однако включение «;» между **end begin** позволит анализатору продолжить работу.

Конечно, в таких ситуациях может иметь место неправильная подстановка, даже если анализатор продолжит работу.

Правила для ошибок

Одним из способов исправления некоторых типов синтаксических ошибок заключается в расширении синтаксиса языка за счет включения в него программ, содержащих типичные ошибки. Это не значит, что ошибки пройдут незамеченными, так как в грамматику могут быть включены сообщения о них. Но анализатор не будет считать такой вход недопустимым и не потребует никаких исправлений. Так можно обращаться, например, с ошибками типа «;» перед **end** или пропуск «;». Дополнительные правила, включенные в грамматику, обычно называются *правилами для ошибок*. Они неизбежно приводят к увеличению грамматики, и поэтому включать их следует только для наиболее часто встречающихся ошибок программирования. При этом надо следить за тем, чтобы при включении этих правил грамматика не стала неоднозначной.

11.6. Предупреждения

Наряду с сообщениями о синтаксических ошибках анализатор может выдавать предупреждения, когда ему встретилась допустимая, но маловероятная последовательность символов, например

; **do**

Еще чаще такие ситуации возникают, когда в таблице идентификаторов содержится переменная, но ссылки в программе на нее нет. Для

выдачи сообщений о таких ситуациях в грамматику вводятся действия, идентифицирующие их.

11.7. Сообщения о синтаксических ошибках

Всякий раз при обнаружении анализатором синтаксической ошибки должно печататься соответствующее сообщение. Например

SYNTAX ERROR IN LINE 22.

Или местоположение ошибки может описываться полнее

LINE 22 SYMBOL 4.

В любом случае пользователь может быть недоволен тем, что сообщение не вполне ясное, так как не указывается, в чем заключается ошибка программиста. На практике фактическая ошибка программирования могла произойти гораздо раньше, анализатор же сообщает об ошибке только тогда, когда ему встречается недопустимый символ. Если программист представляет анализатору программу, имеющую синтаксическую ошибку, компилятор, естественно, не сможет решить, какую программу программист должен был написать. Единственное, что компилятор смог бы сделать, это принять решение о «ремонте» на минимальном расстоянии, т.е. о ремонте, требующем минимальное число включений символов в текст программы и исключений из него, дающем синтаксически правильную программу. Цель ремонта – обеспечить анализатору условия для продолжения анализа программы.

Хотя теоретически ремонт на минимальном расстоянии кажется привлекательным, его реализация неэффективна, так как приходится часто возвращаться назад по уже проанализированным частям программы и отменять выполненные компилятором ранее действия. Большинство компиляторов не берется за такой ремонт. Единственное исправление, которое они осуществляют, - это вставка, исключение или изменение символов *в том месте, где обнаружена ошибка*. В этом случае компилятор не может предоставить иной информации, кроме точного указания о том, где обнаружена ошибка. Компилятору может быть известен еще и контекст, в котором обнаружена ошибка; например, она могла произойти в пределах присвоения, в границах массива или в вызове процедуры. Такая информация не всегда оказывается полезной для пользователя, но она показывает, какой тип конструкции пытался распознать анализатор, когда обнаружил ошибку, а это поможет найти фактическую ошибку программирования. Можно также сообщить пользователю, какие символы допустимы при встрече недопустимого символа. Если анализатор способен сделать разумное предположение о том, какая фактическая ошибка программирования

была допущена, он может исправить программу для последующих проходов.

Для исправления программы (но не ремонта) необходимо знать истинные намерения программиста. В общем случае это невозможно, однако для КС-языков многие типы ошибок можно локализовать достаточно точно.

11.8. Контекстно-зависимые ошибки

Некоторые конструкции типичных языков программирования нельзя описать с помощью контекстно-свободной грамматики. Следовательно, с точки зрения таблицы разбора программы с неописанными идентификаторами синтаксически правильны. Такие контекстно-зависимые ошибки могут быть обнаружены действиями, включаемыми в контекстно-свободную грамматику и вызываемыми анализатором, который запрашивает таблицу символов. Об ошибках такого рода обычно выдаются четкие сообщения при анализе таблицы идентификаторов, например

IDENTIFIER xyz NOT DECLARED
TYPE NOT COMPATIBLE ASSIGNMENT

Так как сам анализатор ошибку не обнаружил, никакого исправления не требуется. Однако, если не принять соответствующие меры, то одна ошибка может повлечь за собой лавину сообщений об ошибках. Во избежание этого при первом же появлении неописанного идентификатора он должен включаться в таблицу символов. В таблицу также должен помещаться тип, соответствующий неописанному идентификатору. Компилятор в этом случае обладает недостаточной информацией, чтобы решить какой тип идентификатора предполагается, поэтому многие компиляторы принимают стандартный тип **int** или **real**. Лучше всего иметь для этого специальный тип **sptype**, который будет ассоциироваться с такими идентификаторами; **sptype** обладает следующими свойствами:

1) его можно приводит к любому типу/виду;
2) если значение типа **sptype** оказывается операндом, знак операции идентифицируется с помощью другого операнда, причем любая неоднозначность разрешается произвольно;
3) применительно к анализатору значение типа **sptype** выбирается или вырезается, хотя при этом выдача соответствующего кода может быть невозможной.

Не исключена и другая ошибка: в одном и том же блоке идентификатор описан дважды. Если возникает такая ситуация, то анализатор обычно генерирует сообщение

IDENTIFIER blank ALREDY DECLARED IN BLOCK

Чтобы избежать неоднозначности, в таблице символов на каждом уровне блоков для каждого идентификатора должен появиться один элемент. Что предпримет компилятор, когда он встречает повторное описание идентификатора в блоке? Оптимальным вариантом было бы проведение во время компиляции подробного анализа части программы, что позволило бы просмотреть, как этот идентификатор используется в блоке, и решить, какое из описаний ему более всего соответствует.

11.9. Ошибки, связанные с употреблением типов

Правила, определяющие, где в программе возможно появление различных значений, не являются контекстно-независимыми. Если идентификатор описан таким образом, что ему могут присваиваться значения в виде целых чисел, то во многих языках попытка присвоить ему литерное значение будет считаться недопустимой.

int *i*; **char** *c*;
i=*c*;

Такая ошибка обнаружится во время компиляции с помощью таблицы символов и выдается сообщение вида

MODE **char** *CANNOT BE COERCED TO* **int.**

Это способствует идентификации ошибки, но вряд ли поможет начинающему программисту.

Особый случай составляют типы, создаваемые программистом, особенно с использованием указателей, поскольку в этом случае в описание одного типа может быть включен другой тип, определяемый пользователем (взаимно рекурсивный тип). Такие ошибки вообще нельзя обнаружить, пока все виды не будут полностью описаны. В этом случае сообщения об ошибках будут выдаваться между проходами компилятора.

Существуют и другие ошибки, связанные с контекстно-зависимыми аспектами типичных языков:

1) неправильное число индексов массива;
2) неправильное число параметров для вызова процедуры или функции;
3) несовместимость типа (или вида) фактического параметра в вызове с типом формального параметра;

4) невозможность определения знака операции по его операндам.

Обычно на такие ошибки компилятор может выдавать четкие сообщения.

11.10. Ошибки, допускаемые во время прогона

Во время прогона в программах могут возникать ошибки, которые нельзя предусмотреть в процессе компиляции. При прогоне могут возникать следующие типы ошибок:

1) нахождение индекса массива вне области действия;
2) целочисленное переполнение (вызванное, например, попыткой сложить два наибольших целых числа, допускаемых реализацией);
3) попытка чтения за пределами файла.

В языках с динамическими типами до времени прогона нельзя обнаружить более широкий класс ошибок (ошибки употребления типов, ошибки, связанные с присвоением и др.)

Обычно компиляторы стараются предотвратить возможность возникновения таких ошибок до прогона. Одно из решений – дать исчерпывающую формулировку задачи, например результат деления на ноль определить как ноль, выходящий за пределы области действия, индекс считать эквивалентным какому-нибудь значению в пределах области действия, при попытке чтения за пределами файла выполнять некоторое стандартное действие и т.д.

Однако такая исчерпывающая формулировка задачи имеет свои «подводные камни»: могут остаться незамеченными ошибки программирования или ошибки данных. Программисты обычно не ожидают, что во время прогона их программ произойдет деление на ноль, - им об этом нужно сообщить. Тем не менее, нежелательно, чтобы из-за этого прерывалось выполнение программы. Компромиссное решение – напечатать сообщение об ошибке времени прогона, когда она возникает, но позволить программе выполнить какие-либо стандартные действия, чтобы она могла продолжать работу и находить дальнейшие ошибки.

В случае ошибки, возникающей в процессе прогона, не всегда можно четко объяснить программисту, что именно неправильно. К этому моменту программа уже транслирована в машинный код, а программисту понятны только ссылки на исходный текст. Поэтому система, работающая при прогоне, должна иметь доступ к таблице идентификаторов и другим таблицам и следить за номерами строк в исходной программе. Таблицы, требуемые для диагностики, к началу прогона могут уже не находиться в основной памяти, но в случае

ошибки должны туда загружаться и фиксировать профиль программы на это время. Эта информация позволяет локализовать место возникновения ошибки или, по крайней мере, блок (рамку), внутри которой возникла аварийная ситуация.

11.11. Ошибки, связанные с нарушением ограничений

Априори программисты предполагают, что компилятор должен быть в состоянии скомпилировать *любую* программу, написанную на исходном языке. Однако это не всегда так из-за конечных технических характеристик конкретной ЭВМ. Хороший компилятор имеет мало произвольных ограничений, но если ограничения вводятся, они должны быть такими, чтобы устраивать подавляющее большинство программ. Обычно в таких случаях вводятся ограничения:

1) на размер программы, которую можно скомпилировать;
2) на число элементов в таблице символов или идентификаторов;
3) на размер стека разбора или других стеков времени компиляции.

Если один и тот же объем памяти отводится под совместное пользование для различных таблиц, то может быть ограничен общий объем, а не объем, занимаемый конкретной таблицей.

Существует вероятность того, что программа заставит нарушить какое-нибудь из ограничений. В этом случае важно, чтобы компилятор выдавал четкое сообщение пользователю, какое именно ограничение он нарушил.

Контрольные вопросы

1. Типы ошибок, возникающие при написании программ.
2. Технология исправления ошибок. Режим переполоха.
3. Технология исправления ошибок. Исключение символов. Включение символов.
4. Правила для ошибок.
5. Предупреждения и сообщения о синтаксических ошибках.
6. Контекстно-зависимые ошибки.
7. Ошибки времени прогона.
8. Ошибки, связанные с нарушениями ограничений.

Список литературы

1. Ахо А., Ульяман Дж. Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1978. - 612 с.
2. Ханкер Р. Проектирование и конструирование компиляторов. – М.: Финансы и статистика, 1984 - 230 с.
3. Райуорд-Смит В.Дж. Теория формальных языков. Вводный курс.-М.: Радио и связь, 1988.
4. Льюис Ф., Розешкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. -М.: Мир, 1979.
5. Вайгартен Ф. Трансляция языков программирования. - М.: Мир, 1977.
6. Гросс М., Лантен А. Теория формальных грамматик. - М.: Мир, 1971.