Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное
образовательное учреждение высшего образования
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

В. Т. Калайда, В. В. Романенко

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Учебное пособие

Томск – 2019

Рецензенты:

Самохвалов И. В., докт. физ.-мат. наук, профессор, зав. кафедрой оптико-электронных систем и дистанционного зондирования Национального исследовательского Томского государственного университета;

Горитов А. Н., докт. техн. наук, профессор кафедры автоматизированных систем управления ТУСУР.

Калайда В.Т.

Теория языков программирования и методы трансляции: учебное пособие / В. Т. Калайда, В. В. Романенко. – Томск: ТУСУР, 2019. – 264 с.

Настоящее пособие посвящено проблеме теоретического описания конечных автоматов, формальных языков и методов трансляции программ. В пособии рассматриваются такие вопросы, как синтаксический и семантический анализ цепочек символов, генерация объектного кода программ (включая оптимизацию кода и исправление ошибок), а также проектирование компиляторов.

Оглавление

Введение

Существует достаточно большое количество вариантов организации трансляции программы, написанной на одном из языков программирования (рис. 1).

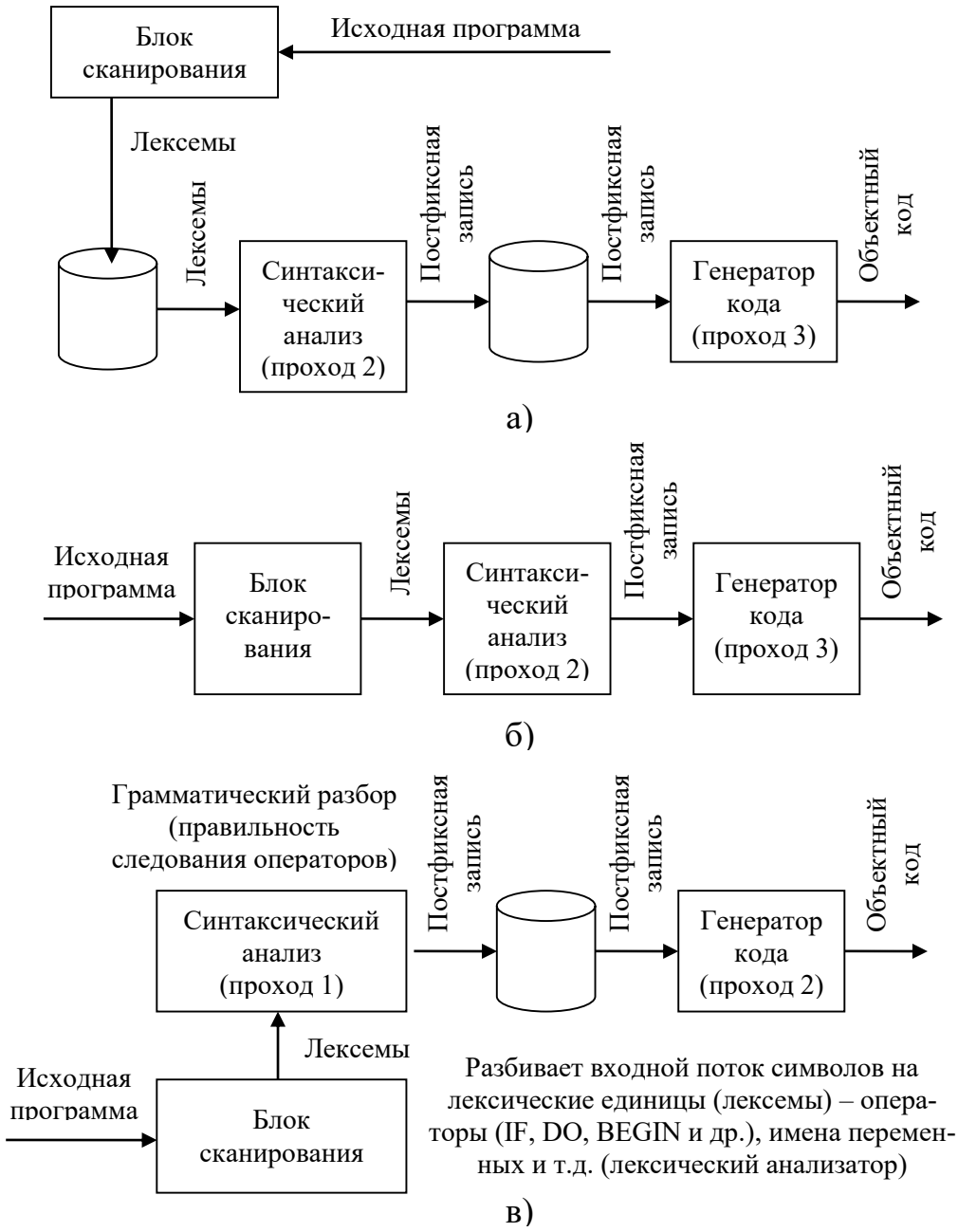


Рисунок 1 – Схемы вариантов организации вычислительного процесса

Однако всем этим схемам присуща общая технологическая цепочка – «лексический анализ → синтаксический анализ → генерация кода → опти-

мизация 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*** [3]. 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 $a \in A$, $A = \{a_1, a_2, \ldots, a_n\}$. Отрицание этого утверждения записывается как $a \notin A$. 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 [3, 5]. 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).

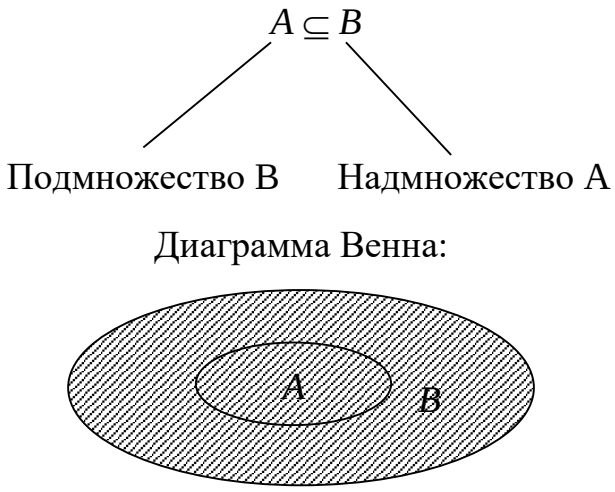


Рисунок 1.1 – Диаграмма Венна для включения множеств

1.2 Операции и отношения

1.2.1 Операции над множествами

Перечислим основные операции над множествами.

1) ***Объединение множеств***

$$A \cup B = \{x \mid x \in A \text{ или } x \in B\} –$$

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

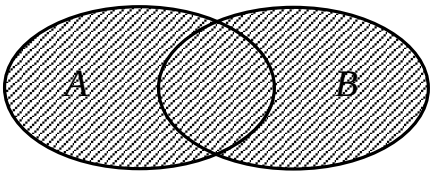


Рисунок 1.2 – Диаграмма Венна для объединения множеств

2) ***Пересечение множеств***

$$A \cap B = \{x \mid x \in A \text{ и } x \in B\} –$$

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

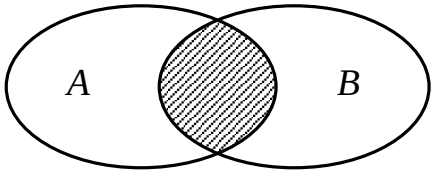


Рисунок 1.3 – Диаграмма Венна для пересечения множеств

Если $A \cap B = \varnothing$, то множества A и B не пересекаются.

3) ***Разность множеств***

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

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

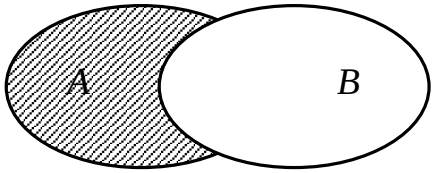


Рисунок 1.4 – Диаграмма Венна для разности множеств

Множество всех элементов, рассматриваемых в данной ситуации, называется ***универсальным множеством*** и обозначается через U.

Разность $U - B = \overline{B}$ называется ***дополнением*** B. Дополнение B и B не пересекаются.

. . . . . .

Пусть I – некоторое множество, элементы которого используются как индексы, и для каждого $i \notin I$ множество A_i известно. Введем обобщенное понятие объединения как

$$\bigcup_{i \in I} A_i = \{X \mid \textit{существует такое } i \in I, \textit{что } X \in A_i\}.$$

. . . . . .

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

Пример

Например, $\bigcup_{i>2} A_i$ означает $A_3 \cup A_4 \cup A_5 \ldots$

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

То есть, если A – конечное множество из m элементов, то $\boldsymbol{P}(A)$ состоит из 2^m элементов.

Пример

Например, пусть $A = \{1, 2\}$. Тогда $\boldsymbol{P}(A) = \{\varnothing, \{1\}, \{2\}, \{1, 2\}\}$.

4) ***Декартово произведение***

В общем случае элементы множества не упорядочены.

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 $\{a, \{a, b\}\}$. Тогда декартово произведение

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

это множество, элементами которого являются всевозможные упорядоченные пары элементов двух исходных множеств.

Пример

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

1.2.2 Отношения на множествах

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

Пусть A – множество целых чисел. Отношение «меньше», которое обозначим как L (aLb), представляет множество $\{(a, b) \mid a < b\}$ или просто $a < b$.

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

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. Это можно записать как $a \equiv b \pmod{N}$, т.е. $a - b = kN$ (k – целое).

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

$$[1] = \{1, 4, 7, \ldots, 3n+1, \ldots\};$$

$$[2] = \{2, 5, 8, \ldots, 3n+2, \ldots\}.$$

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

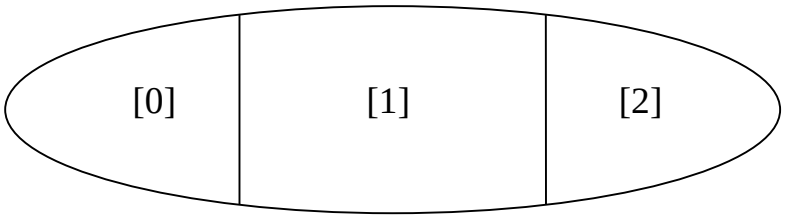
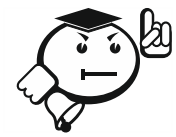


Рисунок 1.5 – Классы эквивалентности отношения сравнения по модулю 3

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



Важным понятием является ***замыкание отношений***. Предположим, стоит задача – как для данного отношения R найти дру-

гое отношение 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.

. . . . . .

Степень отношения R на A обозначается числом k и определяется следующим образом:

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:

$[aR^4b \rightarrow aRc_1$ и $c_1R^3b \rightarrow c_1Rc_2$ и $c_2R^2b \rightarrow c_2Rc_3$ и $c_3R^1b]$.

. . . . . .

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

. . . . . .

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

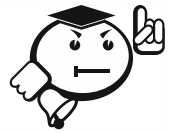
1) aR^*a *для всех* $a \in A$;

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.



Важную роль при изучении алгоритмов играют отношения порядка, особенно специальный вид порядков – ***частичный порядок***.

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.

Другими словами, пусть $S = \{e_1, e_2, \ldots, e_n\}$ – множество, состоящее из n элементов, и пусть $A = \boldsymbol{P}(S)$. Положим aRb для любых a и b из A тогда и только тогда, когда $a \not\subset b$. Тогда отношение R является частичным порядком.

Пример

Для случая $S = \{0, 1, 2\}$ имеем частичный порядок, изображенный на рис. 1.6.

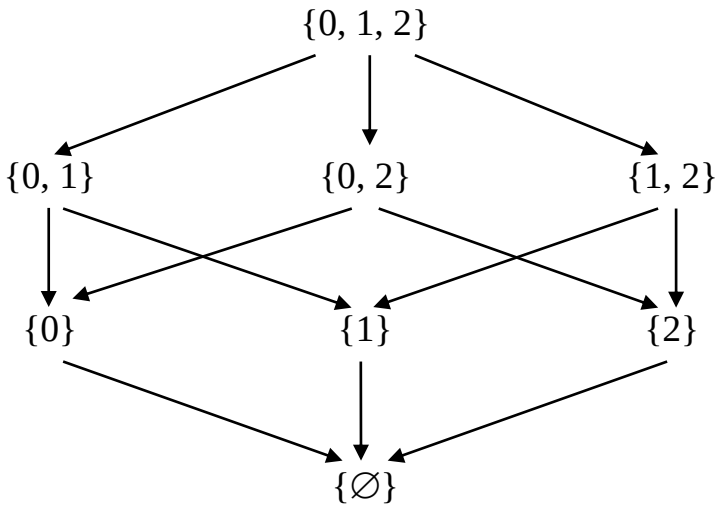


Рисунок 1.6 – Частичный порядок

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. Еще один вид порядков – *линейный порядок*.

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 элементов $a_1, a_2, \ldots, a_n$, для которых $a_i R a_j$ тогда и только тогда, когда $i < j$.*

Аналогично определяется *рефлексивный линейный порядок*.

Пример

Из традиционных систем отношение «<» (меньше) на множестве неотрицательных целых чисел – это линейный порядок, отношение «≤» – рефлексивный линейный порядок.

. . . . . .

И последнее – это *отображение* одних множеств в другие. Отображения также называют *функциями преобразования*.

. . . . . .

***Отображением** 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* $M(a) = b$.

. . . . . .

Отображение $M(a)$ 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.

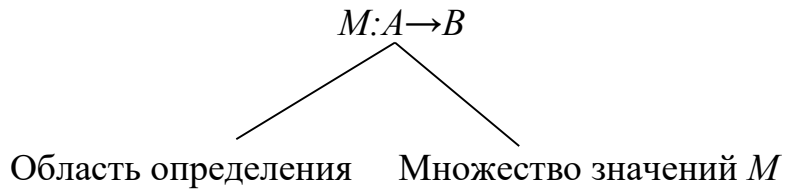


Рисунок 1.7 – Отображение множеств

Если отображение $M{:}A \to B$ таково, что для каждого $b \in B$ существует не более одного $a \in A$, такого, что $M(a) = b$, то M 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}.

. . . . . .

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

. . . . . .

Дадим еще ряд определений:

1) множество *S конечно*, если оно равномощно множеству {1, 2, …, *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

1.3.1 Цепочки

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

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

Особый вид цепочки – пустая цепочка [3, 5], обозначается как *e*. Пустая цепочка не содержит символов.

Соглашения:

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

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

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).

1.3.2 Операции над цепочками

Пусть x, y – цепочки. Тогда:

– Цепочка xy называется *сцепленной* (***конкатенацией*** цепочек x и y). Например, если $x = ab$ и $y = cd$, то $xy = abcd$. Для любой цепочки x можно записать, что $xe = ex = x$.
– ***Обращением*** цепочки x (x^R) называется цепочка x, записанная в обратном порядке: $x = a_1a_2\ldots a_n$, $x^R = a_na_{n-1}\ldots a_1$, $e^R = e$.
– Пусть x, y, z – цепочки в некотором алфавите Σ, тогда, x – ***префикс*** цепочки xy, y – ***суффикс*** цепочки xy, y – ***подцепочка*** 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* $x = a_1a_2\ldots a_n$, *то длина цепочки* n. *Длину цепочки обозначают* $|x|$.

Например:

– $|abc| = 3$;
– $|e| = 0$.

1.4 Языки

1.4.1 Определения

Языком *в алфавите* Σ *называют множество цепочек в* Σ.

Через Σ^* обозначается множество, содержащее все цепочки в алфавите, включая *e* [6].

Пример

Пусть Σ – бинарный алфавит {0,1}, тогда Σ^* = {*e*, 0, 1, 00, 01, 10, 11, 000, 001, …,}.

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

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.

1.4.2 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.

Пусть L_1 – язык в Σ_1, L_2 – язык в Σ_2. Тогда язык L_1L_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^* и определяется следующим образом:

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

$$\bigcup_{n \geq 1} L^n, \text{ т.е. } L^* = L^+ \cup \{e\}.$$

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 $a \in \Sigma_1$.

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 $h(L) = \{a^n b^{2n} \mid n \geq 1\}$.

Если $h : \Sigma_1 \to \Sigma_2^*$, *то отношение*

$$h^{-1} : \Sigma_2^* \to P\left(\Sigma_1^*\right)$$

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

Пусть 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.

1.5.1 Частичные алгоритмы

Можно дать следующее 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 (поиск наибольшего общего делителя двух чисел):

Вход: p и q – положительные целые 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. Величины p и q могут быть очень большими, и, следовательно, затраты на деление будут пропорциональны p и q.

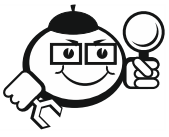
Можно заменить шаг 1 на последовательность шагов, которые вычисляют остаток от деления p на q, причем количество ресурсов, необходимых для выполнения одного такого шага, фиксировано и не зависит от p и q.

. . . . . .

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

1.5.2 Всюду определенные алгоритмы

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 не-

возможен, 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.

Пример

Дан следующий алгоритм:

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;
– оценка 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 $(p, q) – 2q$.

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.

1.5.3 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 язык *L* 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.*

1.5.4 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.

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

1.5.5 Проблемы

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

Например: «целое число *x* меньше целого *y*?».

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

1.6.1 Ориентированные графы

Неупорядоченный ориентированный граф *G – это пара* (*A*, *R*)*, где A – множество элементов, называемых* ***вершинами***, *R – отношения на множестве A. Пара* $(a, b) \in R$ *называется* ***дугой*** (*или* ***ребром***) *графа G.*

Пример

Граф $G = (A, R)$, $A = \{1, 2, 3, 4\}$, $R = \{(1, 1), (1, 2), (2, 3), (2, 4), (3, 4), (4, 1), (4, 3)\}$ (рис. 1.8).

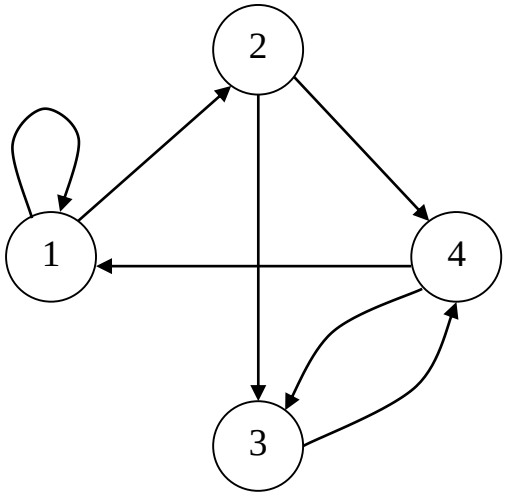


Рисунок 1.8 – Пример ориентированного графа

Пусть $G_1 = (A_1, R_1)$ и $G_2 = (A_2, R_2)$. Графы G_1 и G_2 являются равными (изоморфными), 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 вершин и/или дуг графа иногда приписывают некоторую информацию (разметку). Такие графы называются *помеченными*.

Пусть (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.*

Графы $G_1 = (A_1, R_1)$ и $G_2 = (A_2, R_2)$ являются равными помеченными графами, если существует такое биективное отображение h: $A_1 \rightarrow A_2$, 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

Рассмотрим графы $G_1 = \{\{a, b, c\}, \{(a, b), (b, c), (c, a)\}\}$ и $G_2 = \{\{0, 1, 2\}, \{(1, 0), (2, 1), (0, 2)\}\}$ (рис. 1.9).

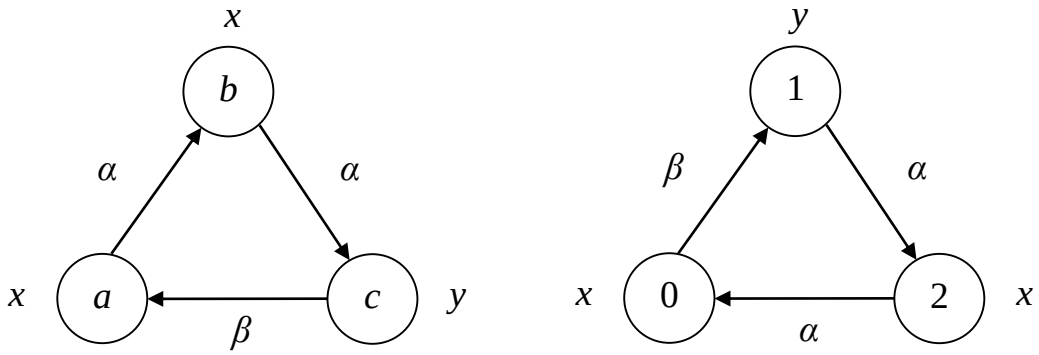


Рисунок 1.9 – Равные помеченные графы

Разметка графа G_1 определяется формулами:

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

$f_1(c) = y$;

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

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

Разметка графа G_2 определяется формулами:

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

$f_2(1) = y$;

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

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

Вывод: графы равны.

. . . . . .

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, **степенью по выходу** – число выходящих из нее дуг.*

. . . . . .

1.6.2 Ориентированные ациклические графы

. . . . . .

***Ациклическим графом** называется граф, не имеющий циклов.*

. . . . . .

. . . . . Пример . . . . . .

Пример ациклического графа приведен на рис. 1.10.

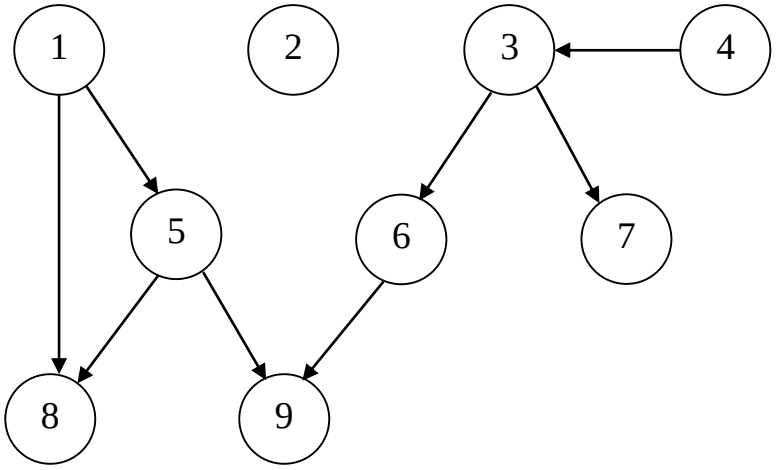


Рисунок 1.10 – Пример ациклического графа

Вершина, степень которой по входу равна 0, называется ***базовой***. Вершина, степень которой по выходу равна 0, называется ***листом*** (или концевой вершиной). Если (a, b) – дуги ациклического графа, то a называется прямым предком b, а b – прямым потомком a.

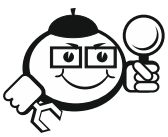
1.6.3 Деревья

Деревом T называется ориентированный граф $G = (A, R)$ со специальной вершиной $r \in A$, 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.

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

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'.

 Пример

Пример дерева и поддерева приведен на рис. 1.11.

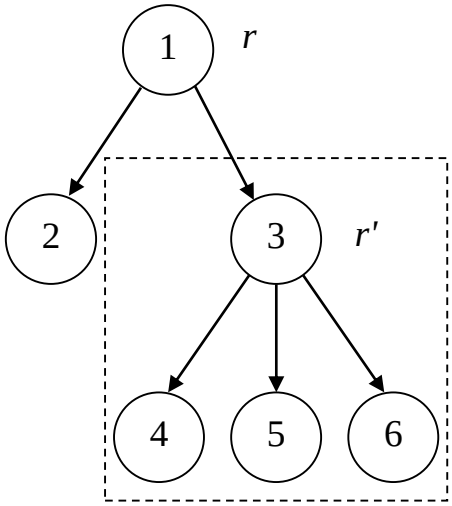


Рисунок 1.11 – Пример дерева и поддерева

1.6.4 Упорядоченные графы

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

Пример

Пример упорядоченного графа приведен на рис. 1.12.

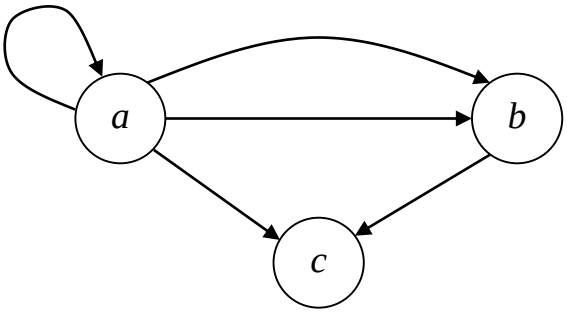


Рисунок 1.12 – Упорядоченный граф

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 $((a, b_1), (a, b_2), \ldots, (a, b_n))$ является последовательность из n символов (помеченные дуги).

Контрольные вопросы по главе 1

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 [1]. Например, если выполняются матричные операции, то для выражения того обстоятельства, что матрица *A* получается перемножением матриц *B* и *C*, удобнее написать команду вида

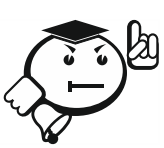
$$A = B*C,$$

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 ее решения 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. То есть 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 исходном 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* вы-

дает 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 – *синтаксического* и *семантического*.

Синтаксическое отображение *связывает с каждым входом* (*программой на исходном языке*) *некоторую структуру, которая служит аргументом семантического отображения.*

Почти всегда структурой любой программы является помеченное дерево. Поэтому сущность алгоритмов перевода обычно сводится к построению подходящих деревьев для входных программ [3].

Пример

В качестве примера того, 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.

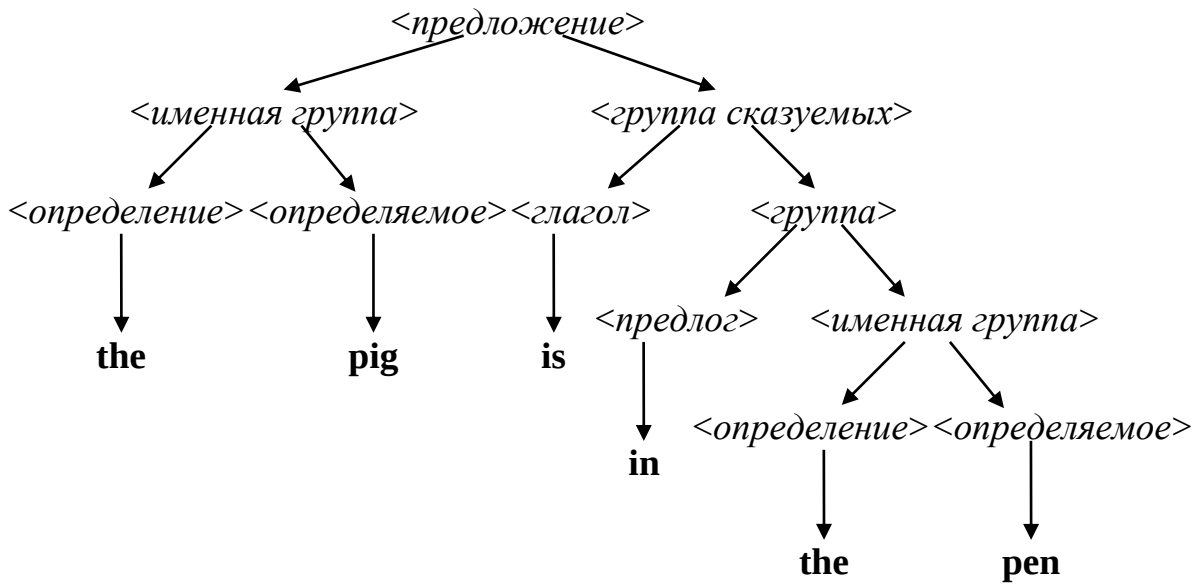
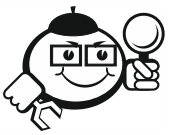


Рисунок 2.1 – Древовидная структура английского предложения

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



Дерево для цепочки *a+b*c* приведено на рис. 2.2.

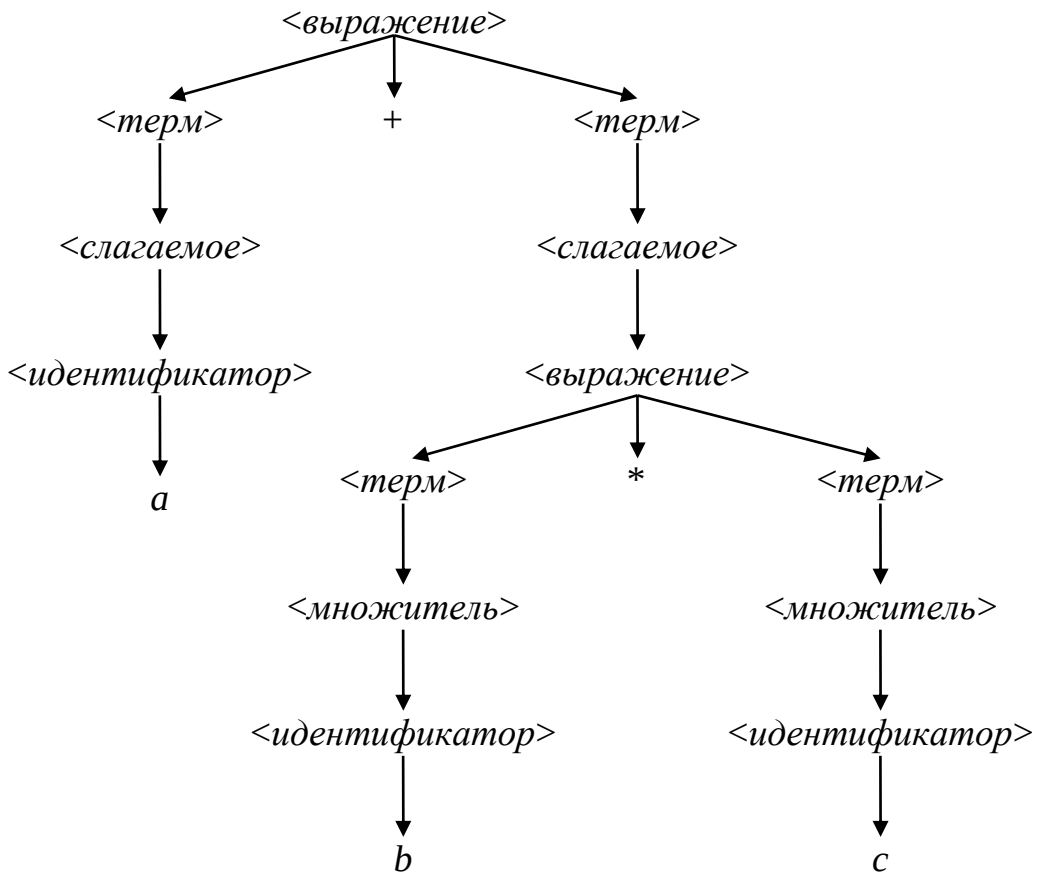


Рисунок 2.2 – Дерево арифметического выражения

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 цепочку 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 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]:

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 *– это 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», второй компонентой обычно является указатель: в ней указывается адрес ячейки, хранящей информацию о конкретной лексеме. Для данного языка число типов лексем считается конечным.

Лексический анализатор *– это транслятор, входом которого служит цепочка символов, представляющая программу, а выходом – последовательность лексем. Этот выход образует вход синтаксического анализатора.*

Пример

Рассмотрим 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 «=», «+» и «*» 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 (типа «*», «+», «=», «(», «)» и т.д.). Однако в общем случае лексический анализ выполнить не так легко.

Пример

Рассмотрим два правильных предложения Фортрана:

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

В первом операторе цепочка **DO** 10 *I* – переменная, а цепочка 1.15 – константа.

Во втором операторе **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, лексический 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 существуют другие подходы к лексическому анализу, менее удобные, но позволяющие избежать проблемы произвольного заглядывания вперед. Отметим 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, образующего 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. Каков характер этой информации – зависит от языка программирования. Например, таблицу, в которой перечислены все идентификаторы вместе с относящейся к ним информацией, называют ***таблицей имен***, ***таблицей идентификаторов*** или ***таблицей символов***.

Вернемся к оператору Фортрана

$$COST = (PRICE+TAX)*0.98.$$

Здесь *COST, PRICE* и *TAX* – переменные с плавающей точкой. Рассмотрим вариант таблицы имен (табл. 2.1).

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 таблице имен, 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 имен 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.

Пример

Синтаксическая структура цепочки *A*+*B***C* должна отражать тот факт, что сначала перемножаются *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

Для цепочки

$$<ИД_1>=(<ИД_2>+<ИД_3>)*<ИД_4>$$

необходимо выполнить следующие действия:

1) $<ИД_3>$ прибавить к $<ИД_2>$;
2) результат (1) умножить на $<ИД_4>$;
3) результат (2) поместить в ячейку, резервированную для $<ИД_1>$.

Этой последовательности соответствует дерево, изображенное на рис. 2.3.

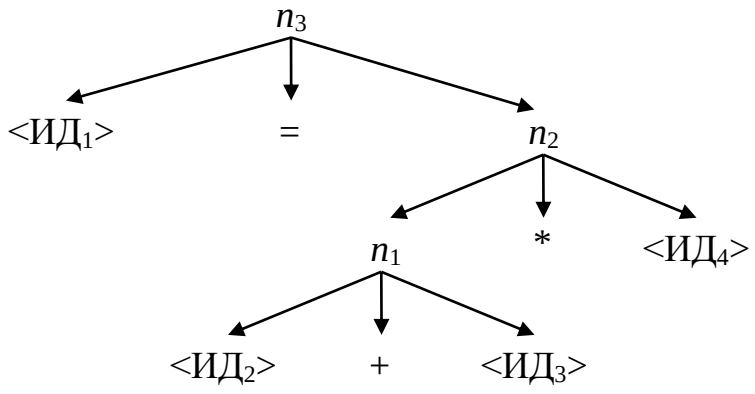


Рисунок 2.3 – Последовательность действий при вычислении выражения

То есть 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 знаки «+», «*» и «=». 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 трехадресный код (из операторов, содержащих не более трех 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 «$C(m)$ → сумматор» означает, что содержимое ячейки памяти m надо поместить в сумматор. Запись $=m$ означает численное значение m.

Таблица 2.2 – Команды языка типа «ассемблер»

Команда	Действие
`LOAD  M`	$C(m)$ → сумматор
`ADD  M`	C(сумматор) + $C(m)$ → сумматор
`MPY  M`	C(сумматор) * $C(m)$ → сумматор

`STORE M`	*C*(сумматор) → *m*
`LOAD =M`	*m* → сумматор
`ADD =M`	*C*(сумматор) + *m* → сумматор
`MPY =M`	*C*(сумматор) * *m* → 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*.

Согласно методу ***синтаксического управляемого перевода****, с каждой вершиной* n *связывается цепочка* C(n) *промежуточного кода. Код для 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 синтаксически управляемой трансляции интерпретация должна задаваться создателем компилятора.

В качестве примера рассмотрим синтаксически управляемую трансляцию арифметических выражений. Допустим, что есть три типа внутренних вершин, зависящих от того, каким из знаков помечен средний потомок: «=», «+» или «*» (рис. 2.4). Здесь треугольники – произвольные поддеревья (в том числе, состоящие из единственной вершины).

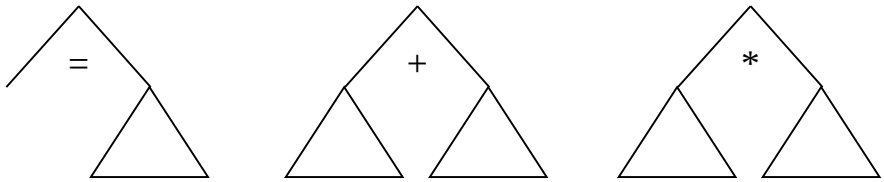


Рисунок 2.4 – Типы вершин

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.


Пример

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 $n_1, n_2, \ldots, n_k$, то

$$l(n) = \max_{1 \le i \le k} l(n_i) + 1.$$

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

Пример

Например, на рис. 2.5 показаны уровни вершин дерева для рассмотренного выше примера.

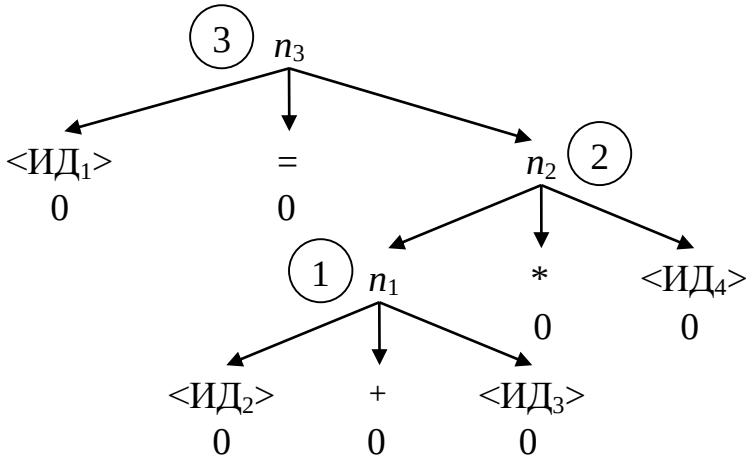


Рисунок 2.5 – Дерево с уровнями

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

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».

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$_i$>.

   1.1. Допустим, что элемент i таблицы идентификаторов является переменной. Тогда $C(n)$ – имя этой переменной.

   1.2. Допустим, что элемент j таблицы идентификаторов является константой k, тогда $C(n)$ – цепочка $=k$.

2) Если n – лист с меткой «=», «+» или «*», то $C(n)$ – пустая цепочка.

3) Если n – вершина типа «а» и ее прямые потомки – это вершины n_1, n_2 и 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 и 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 знак «\$» 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 выражений 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)$.

Пример

Применим этот алгоритм к нашему примеру (рис. 2.6).

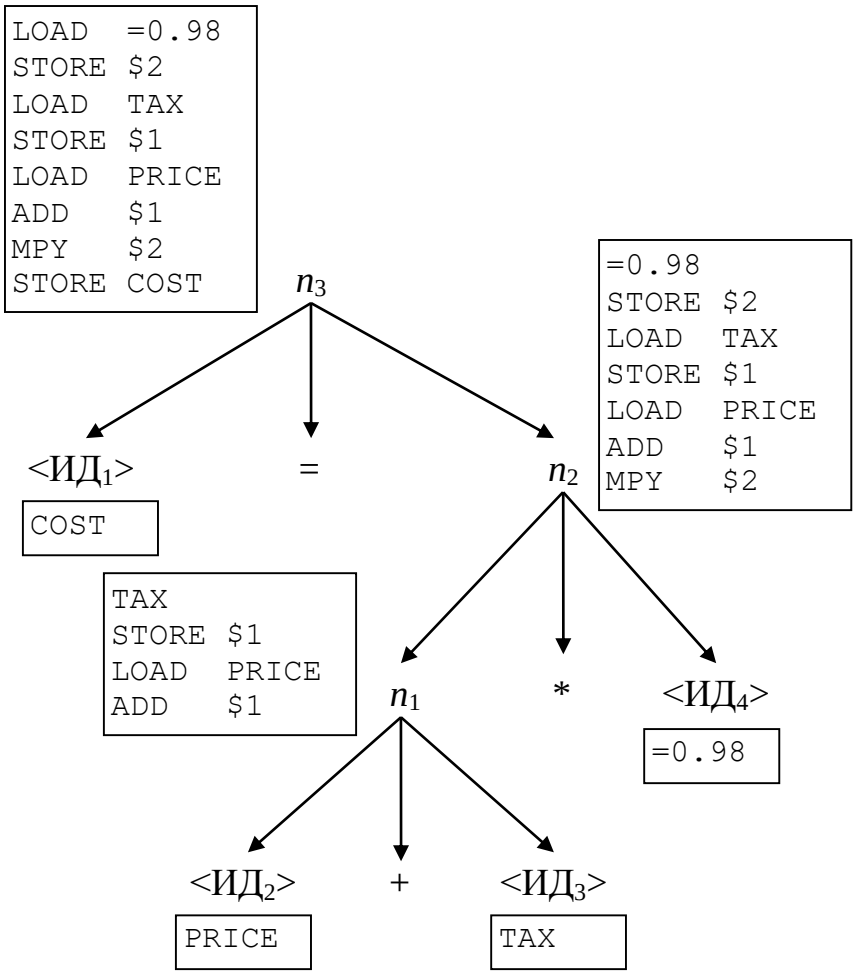


Рисунок 2.6 – Дерево с генерированными кодами

Таким образом, в корне мы получили программу на языке типа «ассемблер», эквивалентную исходной программе на языке Фортран

$$COST = (PRICE+TAX)*0.98.$$

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

2.8 Оптимизация кода

Во многих случаях желательно иметь компилятор, который бы создавал эффективно работающие объектные программы.

*Термином «**оптимизация кода**» обозначаются способы 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, вычисляющим 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 да-

лее. 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 или ячейка α не будет использоваться далее, или 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.

Пример

Рассмотрим наш пример и получим оптимизированную программу (табл. 2.3).

Таблица 2.3 – Оптимизация кода

Этап 1	Этап 2	Этап 3

Применяем правило 1 к последовательности
`LOAD  PRICE`
`ADD  $1`
Заменяем ее последовательностью
	Применяем правило 3 и удаляем последовательность
	`STORE $1`
	`LOAD  $1`
		К последовательности
		`LOAD  =0.98`
		`STORE $2`
		применяем правило 4 и удаляем ее. В команде
		`MPY  $2`

LOAD $1 ADD PRICE		заменяем $2 на =0.98
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

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.

Пример

Например, если $A = B*2C$, то вполне правдоподобно допустить $A = B*2*C$. В общем случае компилятор зафиксирует эту ошибку и остано-

вится. 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 идентификатора месте программы, то компилятор может догадаться, что подразумевается слово INTE**G**ER.
– Вставка одной лексемы, т.е., например, заменить 2*C* на 2**C*.
– Устранение одной лексемы. Например, в операторе **DO** 10 *I*=1,20, – убрать лишнюю запятую.
– Простая перестановка лексем. Например, заменить *I* INTEGER на INTEGER *I*.

Далее мы подробно остановимся на реализации таких компиляторов.

2.10 Резюме

На рис. 2.7 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.

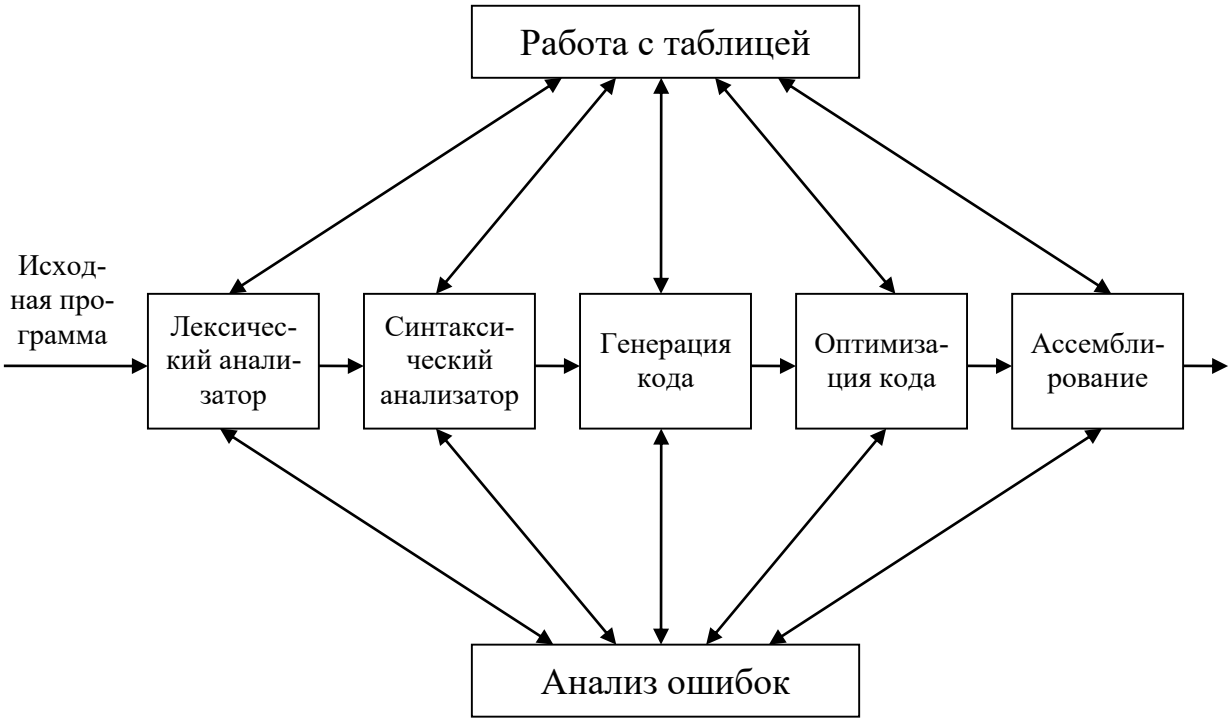


Рисунок 2.7 – Модель компилятора

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

Контрольные вопросы по главе 2

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 Σ [1]. 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 определенным 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).

Другой 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

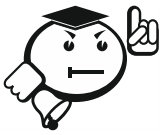
Грамматика *– это математическая система, определяющая вид языка. Одновременно она является устройством, которое придает цепочкам* (*предложениям*) *языка полезную структуру.*

Грамматики образуют наиболее важный класс генераторов языка [1]. 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 определенным способом.*

Сердцевину грамматики составляет конечное множество *P правил образования* (*порождающих правил*), которое описывает процесс порождения цепочек языка.

Правило *– это пара цепочек* (α, β)*, где первой компонентой является любая цепочка, содержащая хотя бы один нетерминал, а второй компонентой – любая цепочка. То есть* $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)^*$*, или, другими словами, правило – это элемент множества декартового произведения* $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$.



Правило (α, β) будем записывать как α→β.

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*.

Грамматикой называется четверка $G = (N, \Sigma, P, S)$, 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 (порождающим правилом или продукцией) и записывается α→β;
- S – выделенный символ из N, называемый начальным (стартовым, исходным) символом.

Пример

Примером грамматики служит четверка $G_1 = (\{A, S\}, \{0, 1\}, P, S)$, где P состоит из правил:

$S \to 0A1$

$0A \to 00A1$

$A \to e$

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

Грамматика определяет язык рекурсивным образом. Рекурсивность проявляется в задании особого рода цепочек, называемых *выводимыми цепочками* грамматики $G = (N, \Sigma, P, S)$, где:

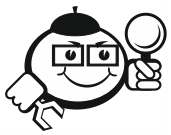
1) S – выводимая цепочка;
2) если $\alpha\beta\gamma$ – выводимая цепочка и $\beta \to \delta$ содержится в P, то $\alpha\delta\gamma$ – тоже выводимая цепочка.

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

Пусть $G = (N, \Sigma, P, S)$ – грамматика. Введем отношение $\Rightarrow_G$ на множестве $(N \cup \Sigma)^*$ ($\varphi \Rightarrow_G \Psi$ означает Ψ, непосредственно выводимая из φ), которое трактуется следующим образом: если $\alpha\beta\gamma$ – цепочка из $(N \cup \Sigma)^*$ и $\beta \to \delta$ – правило из P, то $\alpha\beta\gamma \Rightarrow_G \alpha\delta\gamma$. Транзитивное замыкание отношения $\Rightarrow_G$ обозначается через $\Rightarrow_G^+$ и трактуется как Ψ, выводимая из φ нетривиальным образом. Рефлексивное и транзитивное замыкание отношения $\Rightarrow_G$ ($\Rightarrow_G^*$): $\varphi \Rightarrow_G^* \Psi$ –

означает Ψ, выводимую из φ. Далее, если ясно, о какой грамматике идет речь, то индекс G будет опускаться.

Таким образом, $L(G) = \{w \mid w \in \Sigma^*, S \Rightarrow^* w\}$. Через $\Rightarrow^k$ будем обозначать k-ю степень данного отношения. Иначе говоря, $\alpha \Rightarrow^k \delta$, если существует α_0, α_1, …, α_k, состоящая из $k+1$ цепочки, для которых $\alpha = \alpha_0$, …, α_{i-1}, α_i при $1 \le i \le k$ и $\alpha_k = \beta$. Эта последовательность цепочек называется выводом длины k цепочки β из цепочки α в грамматике G. Отметим, что $\alpha \Rightarrow^* \delta$ тогда и только тогда, когда $\alpha \Rightarrow^i \delta$ для некоторого $i \ge 0$, и $\alpha \Rightarrow^+ \delta$ тогда и только тогда, когда $\alpha \Rightarrow^i \delta$ для некоторого $i \ge 1$.



Пример

Рассмотрим грамматику G_1 из ранее приведенного примера:

$$S \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 0011.$$

На первом шаге S заменяется на $0A1$ в соответствии с правилом $S \to 0A1$. 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 $L(G_1)$.

Набор правил вида:

$\alpha \to \beta_1$

$\alpha \to \beta_2$

……

$\alpha \to \beta_n$

называется *альтернативными порождающими правилами* и обозначается

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

Кроме того, примем еще 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, …, ω 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:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid \text{a}$

Пример вывода в этой грамматике:

$$\begin{aligned} E &\Rightarrow^1 E+T \\ &\Rightarrow^2 T+T \\ &\Rightarrow^3 F+T \\ &\Rightarrow^4 a+T \\ &\Rightarrow^5 a+T*F \\ &\Rightarrow^6 a+F*F \\ &\Rightarrow^7 a+a*F \\ &\Rightarrow^8 a+a*a, \end{aligned}$$

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. Грамматика G называется:

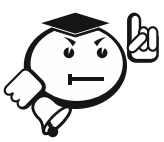
1) *праволинейной*, если $P \subset N \times (\Sigma^+ N \cup \Sigma^*)$, т.е. каждое правило из P имеет вид $A \to xB$, $A \to x$, или $A \to e$, где $A, B \in N$, а x – произвольная цепочка терминалов, $x \in \Sigma^+$;
2) *контекстно-свободной* (или *бесконтекстной*), если $P \subset N \times (N \cup \Sigma)^*$, т.е. 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 $\alpha \to \beta$, $|\alpha| \le |\beta|$, $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)^+$.

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

Пример

Рассмотренный ранее пример (множество арифметических выражений, построенных из символов «*a*», «+», «*» и скобок) является примером контекстно-свободной грамматики.

Заметим, что, согласно введенным определениям, каждая праволинейная грамматика – КС-грамматика. Как видно из определений, контекстно-зависимая грамматика запрещает правила вида $A \to e$ (*e*-правила). Пустая цепочка может входить только в *e*-правило КС-грамматики, т.е. в правило, у которого в правой части находится лишь один символ пустой цепочки *e*.



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

. . . . . .

. . . . . .

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

. . . . . .

3.4 РАСПОЗНАВАТЕЛИ

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

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

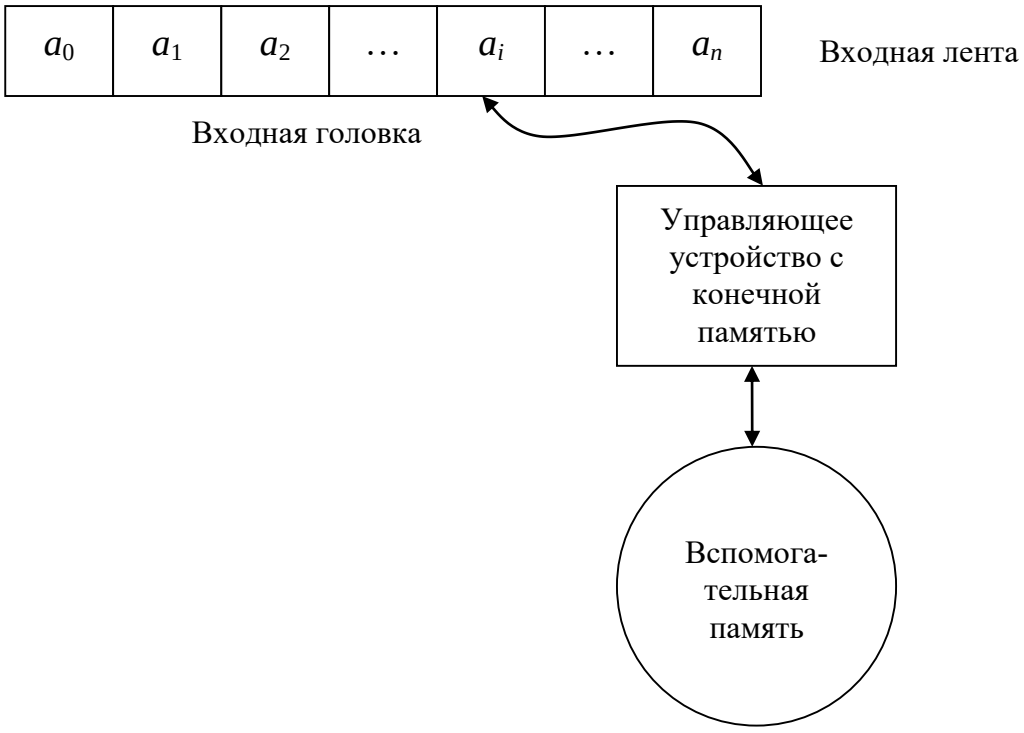


Рисунок 3.1 – Распознаватель

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

Памятью распознавателя может быть хранилище информации любого типа. 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 *функции преобразования памяти*.

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

Функция преобразования памяти *– это отображение, описывающие ее 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, извлеченной 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, извлеченная 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 остается 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. В **недетерминированном** устройстве количество возможных следующих шагов не ограничено.*

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 автоматом 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

3.5.1 Определения

Рассмотрим методы задания языков программирования и класс множеств, образующий этот класс языков. Одним из аппаратов задания являются ***регулярные множества*** и ***регулярные выражения*** на них [1, 3].

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 – регулярные множества 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 ∅, либо $\{e\}$, либо $\{a\}$ для некоторого $a \in \Sigma$, либо его можно получить из этих множеств применением конечного числа операций объединения, конкатенации и итерации.

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

1) ∅ – регулярное выражение, обозначающее регулярное множество ∅;
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 обозначения pp^*. Расстановка приоритетов:

– итерация – наивысший приоритет;
– конкатенация;
– объединение.

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

Пример

Примеры:

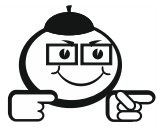
1. 01 означает {01};
2. 0^* – $\{0\}^*$;
3. $(0+1)^*$ – $\{0, 1\}^*$;
4. $(0+1)^*$ 011 – означает множество всех цепочек, составленных из 0 и 1 и оканчивающихся цепочкой 011;
5. $(a+b)$ $(a+b+0+1)^*$ означает множество всех цепочек $\{0, 1, a, b\}^*$, начинающихся с a или b.
6. $(00+11)^*$ $((01+10)(00+11)^*$ $(01+10)(00+11)^*)$ означает множество всех цепочек нулей и единиц, содержащих четное число 0 и четное 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) $\alpha\varnothing = \varnothing\alpha = \varnothing$

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:

$$aa^*b + b = (aa^* + e)b = a^*b,$$

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

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

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

$$X_2 = \alpha_{20} + \alpha_{21}X_1 + \alpha_{22}X_2 + \ldots + \alpha_{2n}X_n$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

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

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

Если $\alpha_{ij} = \varnothing$, то в уравнении для X_i нет члена, 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 системах линейных уравнений.

3.5.2 Алгоритм решения стандартной системы уравнений с регулярными выражениями

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

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:

$$\beta_0 + \beta_{i+1}X_{i+1} + \ldots + \beta_n X_n,$$

причем все β_i – регулярные выражения в алфавите Σ. Затем в правых частях для уравнений $X_{i+1}, \ldots, X_n$ 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 $i = n$).

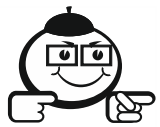
Шаг 5. Уравнение для X_i имеет вид $X_i = \alpha X_i + \beta$, где α и β – регулярные выражения в алфавите Σ. Записать на выходе $X_i = \alpha^*\beta$, в уравнениях для $X_{i-1}, \ldots, X_1$ подставляя $\alpha^*\beta$ вместо X_i.

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

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

является уравнением 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 ситуациях такого рода мы будем брать наименьшее решение, которое назовем *наименьшей неподвижной точкой*. В нашем случае наименьшая неподвижная точка – $\alpha^*\beta$.



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

Пусть Q – стандартная система уравнений с множеством неизвестных $\Delta = \{X_1, X_2, \ldots, X_n\}$ 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 $f: \Delta \rightarrow \boldsymbol{P}(\Sigma^*)$ называется наименьшей неподвижной точкой системы Q, если f – решение, и для любого другого решения g выполняется $f(x) \leq g(x)$ для всех $X \in \Delta$.

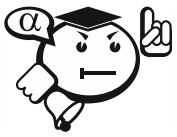
Пусть Q – стандартная система уравнений с неизвестными $\Delta = \{X_1, X_2, \ldots, X_n\}$, и уравнение для X_i имеет вид:

$$X_i = \alpha_{i0} + \alpha_{i1}X_1 + \alpha_{i2}X_2 + \ldots + \alpha_{in}X_n.$$

Тогда наименьшей неподвижной точкой системы Q будет такое отображение:

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

для некоторой последовательности чисел $j_1, j_2, \ldots, j_m$, где $m \geq 1$, $1 \leq k \leq m$, $j_1 = i$.



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, определенные регулярными выражениями;
3) регулярные множества – языки, порождаемые праволинейными грамматиками.

3.6 Недетерминированные конечные автоматы

Еще одним удобным способом определения регулярных множеств являются ***конечные автоматы*** [1, 4]. Качественное описание конечных автоматов мы сделали в разделе 3.4. Теперь дадим их математическую формулировку.

Недетерминированный конечный автомат – это пятерка $M = (Q, \Sigma, \delta, q_0, F)$, где:

- Q – конечное множество состояний;
- Σ – конечное множество допустимых входных символов;
- δ – отображение множества $\Sigma \times Q$ во множество $\boldsymbol{P}(Q)$, 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***.

. . . . . .

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

. . . . . .

. . . . . .

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

. . . . . .

Такт автомата M представляется бинарным отношением $\vdash_M$, определенным 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 которые он может перейти за один шаг.

Запись $C \vdash_M^0 C'$ означает, что $C = C'$, а $C_0 \vdash_M^k C_k$ (для $k \geq 1$) – что существует конфигурация C_1, …, C_{k-1}, такая, что $C_i \vdash_M C_{i+1}$ для всех $0 \leq i \leq k$. Запись $C \vdash_M^+ C'$ означает, что $C \vdash_M^k C'$ для некоторого $k \geq 0$. 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 $L(M)$, определяемым автоматом M, называется множество входных цепочек, допускаемых автоматом M, т.е.

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

Пример

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

Таблица 3.1 – Таблица состояний конечного автомата

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

M допускает все цепочки нулей и единиц, содержащих два стоящих рядом 0. Начальное состояние p можно интерпретировать так: «два стоящих рядом нуля еще не появились и предыдущий символ не был нулем». Состояние q означает, что «два стоящих рядом нуля еще не появились, но предыду-

щий символ был нулем». Состояние r означает, что «два стоящих рядом нуля уже появились», т.е. попав в состояние r, автомат остается в этом состоянии.

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

$$(p, 01001) \vdash^1 (q, 1001)$$

$$\vdash^2 (p, 001)$$

$$\vdash^3 (q, 01)$$

$$\vdash^4 (r, 1)$$

$$\vdash^5 (r, e).$$

Таким образом, $01001 \in L(M)$.

Пример

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 состоянии 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	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_3\}$
Состояние q_1	$\{q_1, q_f\}$	$\{q_1\}$	$\{q_1\}$
Состояние q_2	$\{q_2\}$	$\{q_2, q_f\}$	$\{q_2\}$
Состояние q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_f\}$
Состояние q_f	$\varnothing$	$\varnothing$	$\varnothing$

Формально автомат M определяется как пятерка:

$$M = (\{q_0, q_1, q_2, q_3, q_f\}, \{1, 2, 3\}, \delta, q_0, \{q_f\}).$$

3.7 Графическое представление конечных автоматов

Часто удобно графическое представление конечного автомата. Пусть $M = (Q, \Sigma, \delta, q_0, F)$ – недетерминированный конечный автомат.

Диаграммой переходов (*графом переходов*) δ *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* $q \in \delta(p, a)$.

Пример

Изобразим автоматы из предыдущих примеров в виде графов (рис. 3.2 и 3.3).

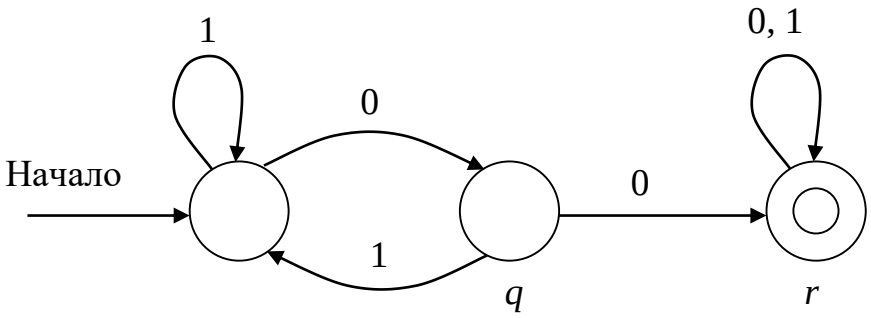


Рисунок 3.2 – Пример детерминированного графа

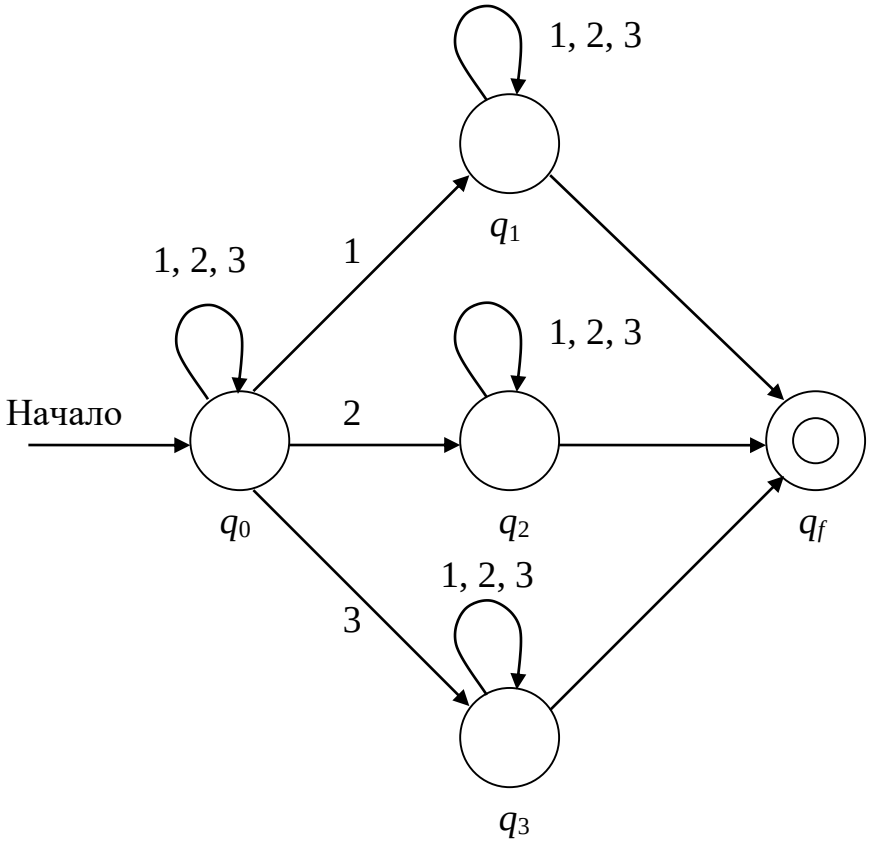


Рисунок 3.3 – Пример недетерминированного графа

. . . . . .

Для дальнейшего анализа нам потребуется определение детерминированного автомата.

. . . . . .

Пусть $M = (Q, \Sigma, \delta, q_0, F)$ *– недетерминированный конечный автомат. Назовем автомат* M ***детерминированным****, если множество* $\delta(q, a)$ *содержит не более одного состояния для лю-*

бых $q \in Q$ *и* $a \in \Sigma$. *Если* $\delta(q, a)$ *всегда содержит точно одно состояние, то автомат* M *назовем* ***полностью определенным***.

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

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) $Q' = \boldsymbol{P}(Q)$, 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 $q \in Q\}$.

Пример

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

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

Начнем 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.

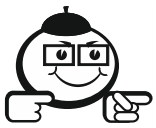
Начальным состоянием автомата M является A, а множество заключительных состояний – $\{E, H, J, K, M, N, P\}$.

. . . . . .

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 [3, 6]. Это следует из следующих лемм, принимаемых нами без доказательств.

. . . . . .



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

. . . . . .

. . . . . .

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

. . . . . .

. . . . . .

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.

. . . . . .

. . . . . .

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

3.9.1 Постановка задачи

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 берется один представитель в качестве состояния сокращенного (или приведенного) автомата [4].

Таким образом, можно сократить объем автомата M, если M содержит недостижимые состояния или два или более неразличимых состояний. Полученный автомат будет наименьший из конечных автоматов, распознающий регулярное множество, определяемое первоначальным автоматом M.

. . . . . .

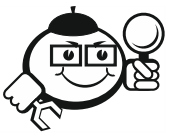
Пусть $M = (Q, \Sigma, \delta, q_0, F)$ *– конечный автомат, а* q_1 *и* q_2 *– два его различных состояния. Будем говорить, что цепочка* $x \in \Sigma^*$ ***различает*** *состояния* q_1 *и* q_2*, если* $(q_1, x) \vdash^* (q_3, e)$, $(q_2, x) \vdash^* (q_4, e)$ *и одно из состояний* q_3 *и* q_4 *принадлежит* F*. Будем говорить, что* q_1 *и* q_2 ***k-неразличимы****, и писать* $q_1 \equiv^k q2$*, если не существует такой цепочки* x*, различающей* q_1 *и* q_2*, у которой* $|x| \le k$*. Будем го-*

ворить, что состояния q_1 *и* q_2 ***неразличимы****, и писать* $q_1 \equiv q_2$*, если они k-неразличимы для любого* $k \geq 0$.

. . . . . .

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, изображенной на рис. 3.4.

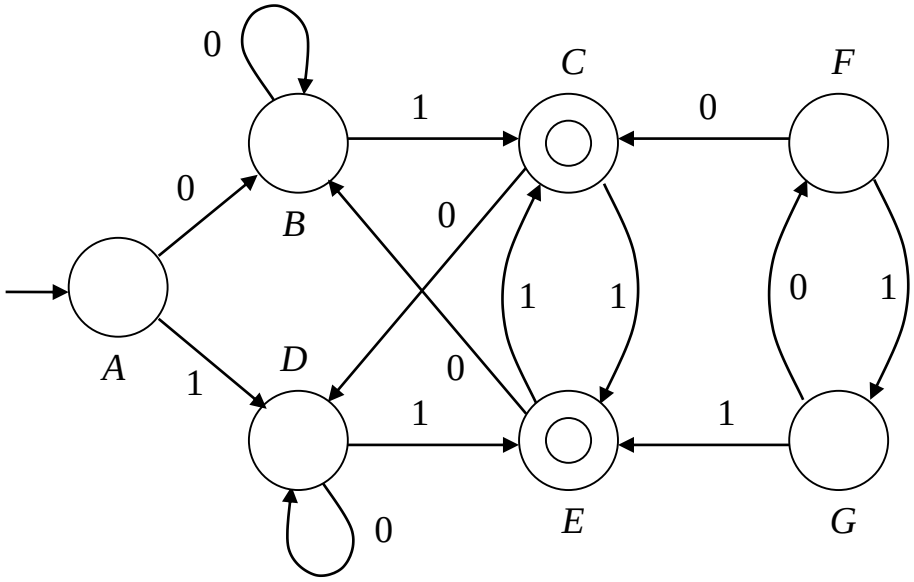


Рисунок 3.4 – Диаграмма автомата

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).

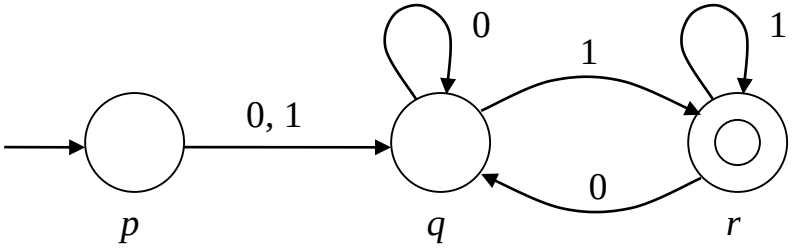


Рисунок 3.5 – Приведенный конечный автомат

Перед тем, как построить алгоритм канонического конечного автомата, введем лемму.

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 $(n–2)$-неразличимы, т.е. если два состояния можно различить, то их можно различить с помощью входной цепочки, длина которой меньше числа состояний автомата.

3.9.2 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' – это $\{[q] \mid q \in F\}$.

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

Пример

Найдем приведенный конечный автомат для автомата M, изображенного на рис. 3.6.

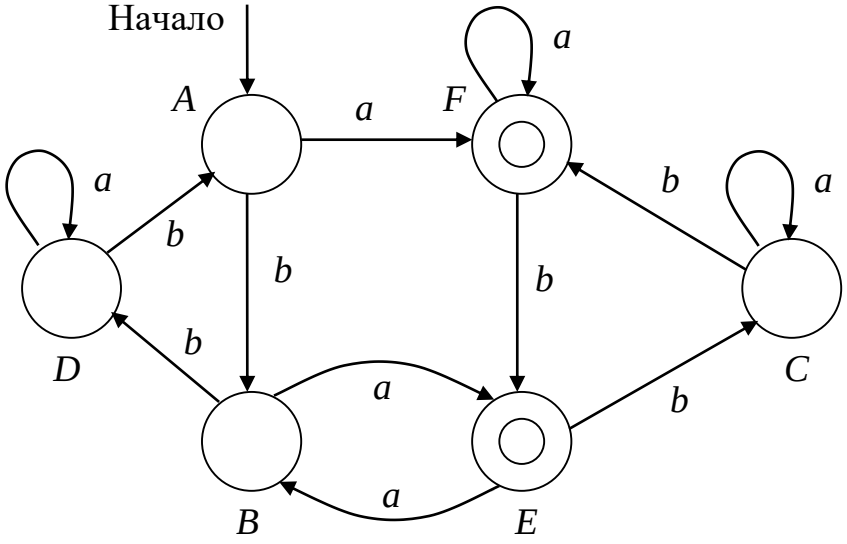


Рисунок 3.6 – Диаграмма автомата

Отношения ≡ для $k \geq 0$ 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 $\equiv = \equiv^1$. Приведенный автомат M' будет ({[A], [B], [C], $\{a, b\}$, δ', A, {[A]}}), где δ' определяется следующей табл. 3.4.

Таблица 3.4 – Приведенный конечный автомат

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

Здесь:

– [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 четырех 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 [1].

Синтаксическую структуру входной цепочки можно определить по последовательности правил, применяемых при выводе этой цепочки. Таким об-

разом, 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 [5].

Дерево вывода в КС-грамматике $G = (N, \Sigma, P, S)$ – это помеченное упорядоченное дерево, каждая вершина которого помечена символом из множества $N \cup \Sigma \cup \{e\}$, где:

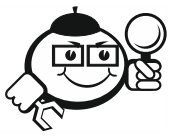
- N – конечное множество нетерминальных символов;
- Σ – непересекающееся с N множество терминальных символов;
- P – конечное подмножество множества $N \times (N \cup \Sigma)^*$ (элемент (A, α) множества P называется правилом или продукцией $A \to \alpha$);
- S – выделенный символ из N, называемый начальным или исходным символом.

Если внутренняя вершина помечена символом A, а ее прямые потомки – символами $X_1, X_2, \ldots, X_n$, то $A \to X_1 \mid X_2 \mid \ldots \mid X_n$ – правило грамматики.

Помеченное упорядоченное дерево D называется деревом вывода (или деревом разбора) в КС-грамматике $G(A) = (N, \Sigma, P, A)$, 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 $A \to X_1 \mid X_2 \mid \ldots \mid X_n$ – правило из множества P. D_i должно быть деревом вывода в грамматике $G(X_i) = (N, \Sigma, P, X_i)$, 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 $S \to aSbS \mid bSaS \mid e$ (рис. 3.7). Возможные варианты деревьев вывода:

$$S \Rightarrow^1 aSbS \Rightarrow^2 abSaSb \Rightarrow^3 abab$$

$$S \Rightarrow^1 bSaS \Rightarrow^2 babSaS \Rightarrow^3 baba$$

$$S \Rightarrow^1 e$$

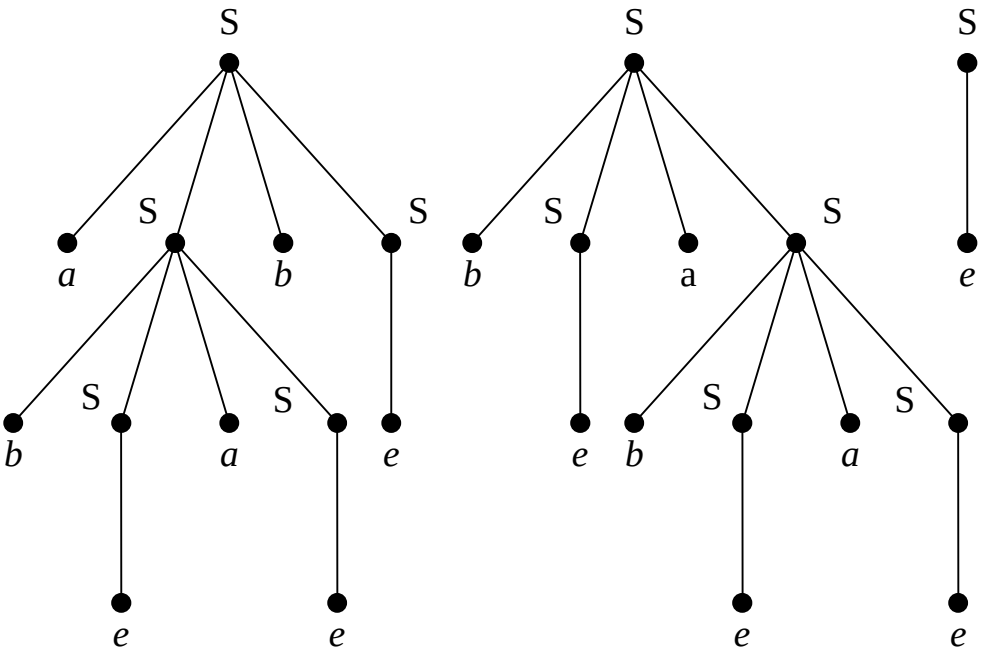


Рисунок 3.7 – Деревья выводов

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 – ее прямые потомки. Тогда, если $i < j$, то вершина n_i и все ее потомки считаются расположенными левее вершины n_i и всех ее потомков.

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

Пример

Например, кроны деревьев на рис. 3.7 – *abab*, *baba*, *e*.

Сечением дерева *D* назовем 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).

Частные случаи:

– множество вершин дерева, состоящего из одного корня, является сечением;
– листья также образуют сечение.

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

Пример

Крона сечения дерева, приведенного на рис. 3.8 – *aba*S*b*S.

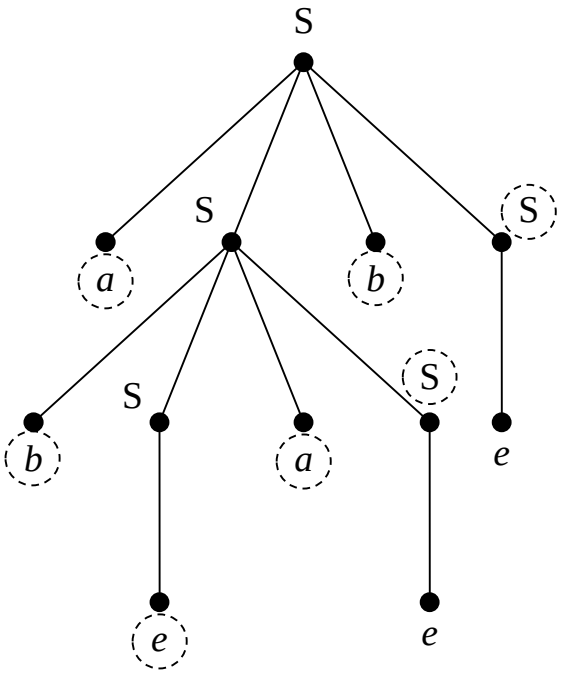
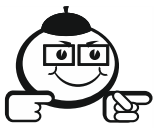


Рисунок 3.8 – Пример сечения дерева

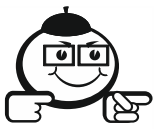
. . . . . .



. . . . . .

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

. . . . . .



. . . . . .

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

. . . . . .

Доказательство. Пусть $C_0, C_1, \ldots, C_n$ – 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 ее 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* $\alpha_1, \alpha_2, \ldots, \alpha_n$ *называется* ***левым выводом*** *цепочки* α_n *из* α_0 *в грамматике* G. ***Правый вывод*** *определяется аналогично.*

Если $S = \alpha_1, \alpha_2, \ldots, \alpha_{n-1} = w$ – 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:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid a.$

Нарисуем дерево вывода (рис. 3.9).

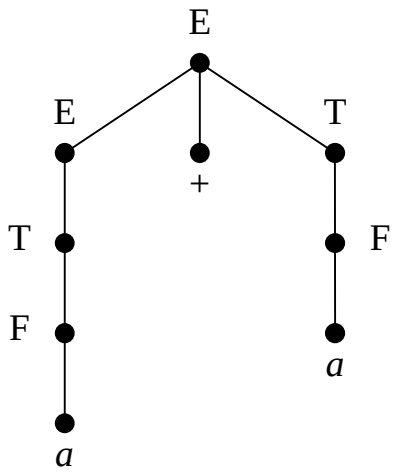


Рисунок 3.9 – Пример дерева вывода

Это дерево служит представлением десяти эквивалентных выводов цепочки $a + a$:

– левый вывод $E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + F \Rightarrow a + a$;

– правый вывод $E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + a \Rightarrow T + \text{a} \Rightarrow F + a \Rightarrow a + a$.

. . . . . .

Цепочку α будем называть *левовыводимой*, если существует левый вывод $S = \alpha_1, \alpha_2, \ldots, \alpha_n = \alpha$, и писать $S \Rightarrow_l^* \alpha$. Аналогично, α будем называть *правовыводимой*, если существует правый вывод $S = \alpha_1, \alpha_2, \ldots, \alpha_n = \alpha$, и писать $S \Rightarrow_r^* \alpha$. 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.

. . . . .  Пример . . . . . .

Рассмотрим, например, язык $L(G_0)$. Этот язык порождается грамматикой G с правилами

$$E \to E{+}E \mid E{*}E \mid (E) \mid a.$$

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

$$E \to E+T \mid E*T \mid T$$

$$T \to (E) \mid a.$$

Другой недостаток грамматики G, которым обладает и грамматика G_1, заключается в том, что операции «+» и «*» имеют один и тот же приоритет. То есть структура выражения $a+a*a$ и $a*a+a$, которую мы придаем 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. Начнем с очевидных, но важных преобразований.

. . . . .  Пример . . . . . .

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 $L(G)$.

. . . . . .

Назовем 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 *и* 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 $\{w \mid A \Rightarrow^* w,\ w \in \Sigma^*\}$.

3.10.2.1. Алгоритм проверки пустоты языка

Алгоритм: Не пуст 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, … рекурсивно.

1. Положить $N_0 = \varnothing$, $i = 1$.
2. Положить $N_i = \{A \mid A \to \alpha \in P \text{ и } \alpha \in (N_{i-1} \cup \Sigma)^*\} \cup N_{i-1}$.
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).

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

3.10.2.2. Алгоритм устранения недостижимых символов

Символ $X \in (N \cup \Sigma)$ *назовем* ***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.

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

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

1. $L(G') = L(G)$;
2. 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 $V_i = \{X \mid$ в P есть $A \to \alpha X \beta$ и $A \in V_{i-1}\} \cup V_{i-1}$.
3) Если $V_i \neq V_{i-1}$, положить $i = i+1$ и перейти к шагу 2), в противном случае, 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. $V_i \subseteq N \cup \Sigma$.

3.10.2.3. Алгоритм устранения бесполезных символов

На базе двух рассмотренных алгоритмов построим обобщенный 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 $S \Rightarrow^* wXy \Rightarrow^* wxy$.

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

3.10.2.4. Алгоритм преобразования в грамматику без *e*-правил

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

Назовем 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 устранения 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 $N_e = \{A \mid A \in N \text{ и } A \Rightarrow^+_G e\}$.
2) Построить P' следующим образом:

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

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

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

   б) если $S \in N_e$, включить в P' правила $S' \to e \mid S$, где S' – новый символ, и положить $N' = N \cup \{S'\}$, в противном случае положить $N' = N$ и $S' = S$.
3) Положить $G' = (N', \Sigma', P', S')$.

Пример

Рассмотрим грамматику $S \to aSbS \mid bSaS \mid e$. Применяя к ней рассмотренный алгоритм, получаем грамматику

$$S' \to S \mid e$$

$S \to aSbS \mid bSaS \mid aSb \mid abS \mid ab \mid bSa \mid baS \mid ba.$

. . . . . .

3.10.2.5. Алгоритм устранения цепных правил

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

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

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

Метод:

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

   а) положить $N_0 = \{A\}$ и $i = 1$;

   б) положить $N_i = \{C \mid B \to C$ принадлежит P и $B \in N_{i-1}\} \cup N_{i-1}$;

   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 $B \to \alpha$ принадлежит P и не является цепным правилом, включать в P' правило $A \to \alpha$ для таких A, что $B \in N_A$.

3) Положить $G' = (N', \Sigma', P', S)$.

. . . . .  Пример . . . . . .

Грамматика с правилами:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid a$

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 таким:

$E \to E+T \mid T*F \mid (E) \mid a$

$T \to T*F \mid (E) \mid a$

$F \to (E) \mid a$

. . . . . .

3.10.3 Грамматика без циклов

. . . . . .

*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 **приведенной**, если она без циклов, без e-правил и без бесполезных символов.*

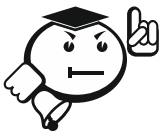
. . . . . .

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

3.10.4 Нормальная форма Хомского

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 [3, 6]:

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



. . . . . .

Аврам Ноам (Наум) Хомский – лингвист, автор работ о порождающих грамматиках и классификации формальных языков, называемой иерархией Хомского.

. . . . . .

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 $A \to X_1X_2\ldots X_k$, где $k > 2$, включить в P' правила

$$A \to X'_1\langle X_2\ldots X_k\rangle$$
$$\langle X_2\ldots X_k\rangle \to X'_2\langle X_3\ldots X_k\rangle$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$\langle X_{k-2}X_{k-1}X_k\rangle \to X'_{k-2}\langle X_{k-1}X_k\rangle$$
$$\langle X_{k-1}X_k\rangle \to X'_{k-1}X'_k,$$

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

5) Для каждого правила из P вида $A \to X_1X_2$, где хотя бы один из символов X_1 и X_2 принадлежит Σ, включить в P' правило $A \to X'_1X'_2$.
6) Для каждого нетерминала вида A', введенного на шагах 4) и 5), включить в P' правило $A' \to a$. Наконец, пусть N' – это N вместе со всеми нетерминалами, введенными при построении P'. Тогда искомая грамматика $G = (N', \Sigma', P', S)$.

Пример

Пусть G – приведенная грамматика, определенная 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', состоящую 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 форма 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 [3].

Нетерминал A в КС-грамматике $G = (N, \Sigma, P, S)$ называется ***рекурсивным****, если $A \Rightarrow^{+} \alpha A \beta$ для некоторых α и β. Если $\alpha = e$,*

то A называется ***леворекурсивным****. Аналогично, если* $\beta = e$*, то A называется* ***праворекурсивным****.*

. . . . . .

. . . . . .

Грамматика, имеющая хотя бы один леворекурсивный терминал, называется ***леворекурсивной****. Аналогично определяется и* ***праворекурсивная*** *грамматика. Грамматика, в которой все нетерминалы, 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.

Пусть $G = (N, \Sigma, P, S)$ – КС-грамматика, в которой

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

т.е. все правила из P вида $A\alpha_i$ содержат левую рекурсию, и ни одна из цепочек β_i не начинается с A.

Если грамматика $G' = (N \cup \{A'\}, \Sigma, P', S)$, где A' – новый нетерминал, а P' получено из P заменой A-правил правилами:

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

$$A' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A',$$

то $L(G') = L(G)$.

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

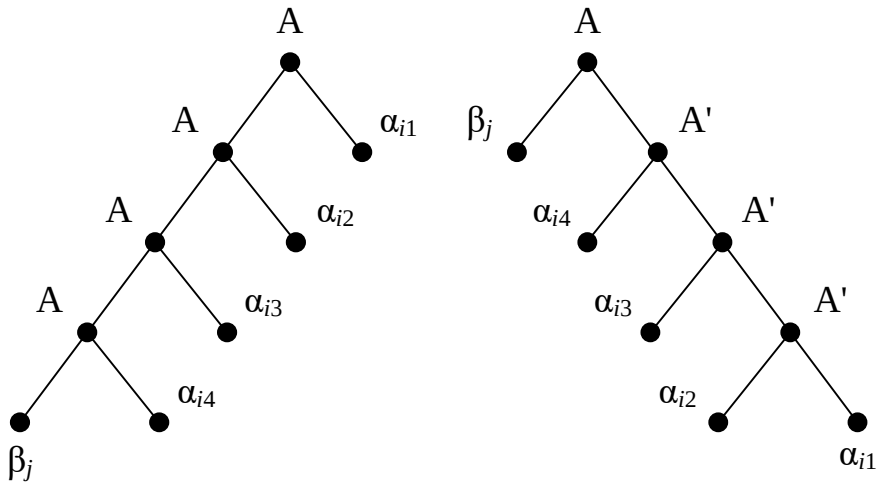


Рисунок 3.10 – Леволинейная и праволинейная и грамматики

Пример

Пусть G_0 – наша обычная грамматика с правилами:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid a$

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

$E \to T \mid TE'$

$E' \to +T \mid +\ TE'$

$T \to F \mid FT'$

$T' \to *F \mid *FT'$

$F \to (E) \mid a$

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

Алгоритм устранения левой рекурсии:

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

Выход. Эквивалентная КС-грамматика без левой рекурсии.

Метод:

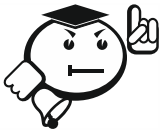
1) Пусть $N = \{A_1, A_2, \ldots, A_n\}$. Преобразуем G так, чтобы в правиле $A_i \to \alpha$ цепочка α начиналась либо с терминала, либо с такого A_j, что $i > j$. С этой целью положим $i = 1$.
2) Во множестве правил $A_i \to A_i\alpha_i \mid \ldots \mid A_i\alpha_m \mid \beta_1 \mid \ldots \mid \beta_p$, где ни одна из цепочек β_i не начинается с A_k, если $k \le i$, заменим A_i-правила правилами:

$$A_i = \beta_1 \mid \ldots \mid \beta_p \mid \beta_1 A_i' \mid \ldots \mid \beta_n A_i' \mid,$$
$$A_i' = \alpha_1 \mid \ldots \mid \alpha_m \mid \alpha_1 A_i' \ldots \mid \alpha_m A_i',$$

   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 $A_i \to \beta_1\alpha \mid \ldots \mid \beta_m\alpha$, где $A_j \to \beta_1 \mid \ldots \mid \beta_m$ для всех A_j-правил. Так как правая часть каждого A_j-правила начинается уже с терминала или A_k для $k > j$, 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.

КС-грамматика $G = (N, \Sigma, P, S)$ 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 $a \in \Sigma$, $\alpha \in N^*$.*



Шейла Адель Грейбах является исследователем формальных языков в области вычислений, автоматов, теории компиляторов и информатики.

3.11 Автоматы с магазинной памятью

Автоматы с магазинной памятью являются естественной моделью синтаксического анализатора КС-языков [1, 3, 4].

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.*

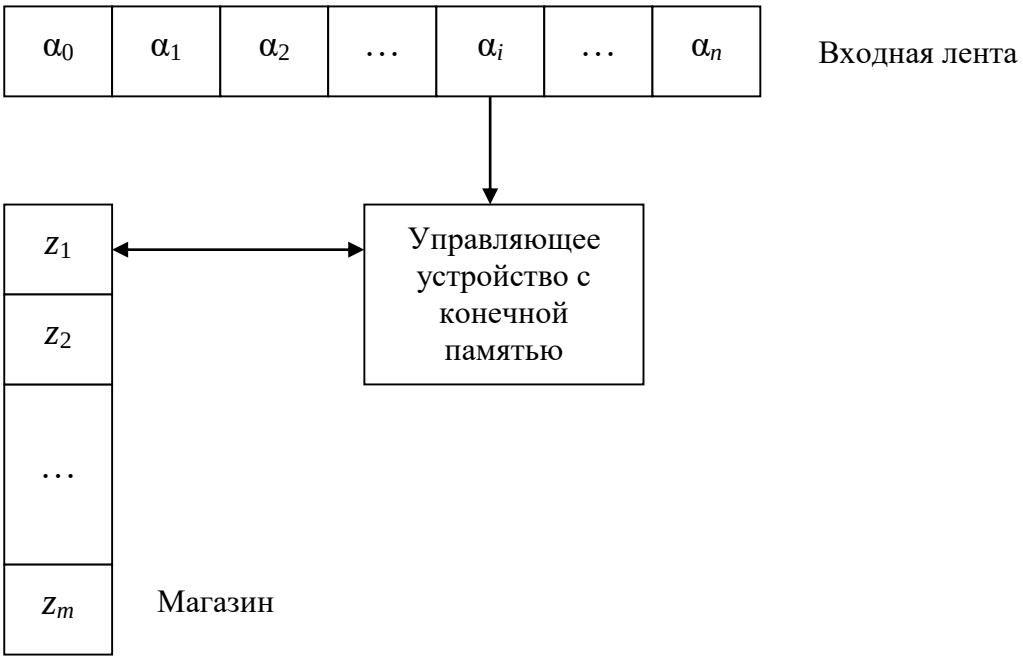


Рисунок 3.11 – Автомат с магазинной памятью

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

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

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

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 $Q\times\Gamma^*$;
- $q_0\in Q$ – 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 $q, q' \in Q$, $a \in \Sigma \cup \{e\}$, $w \in \Sigma^*$, $Z \in \Gamma$ и $\alpha, \gamma \in \Gamma^*$.

Если $\delta(q, a, Z) = (q', \gamma)$, 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;
– 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 $\alpha \in \Gamma^*$. А $L(P)$ – т.е. *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.

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 $w = a_1 a_2 \ldots a_k$ $(k \geq 0)$, то

$$(q_1, a_1 \ldots a_k, A) \vdash (q, a_1 \ldots a_k, a_1 \ldots a_k) \vdash^k (q, e, e).$$

Теперь предположим, что $A \Rightarrow^m w$ для некоторого $m > 1$. Первый шаг этого вывода должен иметь вид $A \Rightarrow X_1X_2\ldots X_k$, где $X_i \Rightarrow^{m_i} x_i$ для некоторого $m_i < m$, $1 \le i \le k$ и $x_1x_2\ldots x_k = w$. Тогда $(q, w, A) \vdash (q, w, X_1X_2\ldots X_k)$. 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 $(q, w, A) \vdash (q, w, X_1X_2\ldots X_k)$, причем $(q, x_i, X_i) \vdash^{n_i} (q, e, e)$ для $1 \le i \le k$ и $x_1x_2\ldots x_k = w$. Тогда $A \Rightarrow X_1X_2\ldots X_k$ – 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,

$$
\begin{aligned}
&A \Rightarrow X_1\ldots X_k\\
&\Rightarrow^* x_1X_2\ldots X_k\\
&\Rightarrow^* x_1x_2X_3\ldots X_k\\
&\ldots\ldots\ldots\ldots\\
&\Rightarrow^* x_1x_2\ldots x_{k-1}X_k\\
&\Rightarrow^* x_1x_2\ldots x_{k-1}x_k = w
\end{aligned}
$$

– вывод цепочки w из A в грамматике G.

Контрольные вопросы по главе 3

1. Способы определения языков.
2. Грамматики.

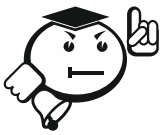
3. Грамматики с ограничениями на правила.
4. Распознаватели.
5. Регулярные множества, их распознавание и порождения.
6. Алгоритм решения системы линейных выражений с регулярными выражениями.
7. Регулярные множества и конечные автоматы.
8. Проблема разрешимости.
9. Графическое представление конечных автоматов.
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 – КС-грамматиками [3, 4, 5].

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

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



Так как ДМП-автоматы содержат 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 описываются КС-грамматиками. Те, в свою очередь, являются частным случаем *s*-грамматик [2, 6], представляющих собой грамматики, в которых:

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

Даны правила грамматики:

$S \to pX$

$S \to qY$

$X \to aXb$

$X \to x$

$Y \to aYd$

$Y \to y$

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 – Разбор строки

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

КС-грамматику $G = (N, \Sigma, P, S)$ можно задавать лишь множеством правил при условии, что правила со стартовым символом в левой части записаны первыми. Тогда:

- Нетерминалами N будут те элементы грамматики, которые встречаются слева от знака вывода в порождающих правилах;
- Терминалами Σ будут все остальные элементы грамматики, за исключением символа пустой цепочки e;
- Стартовым символом грамматики S будет нетерминал из левой части первого порождающего правила.


Пример

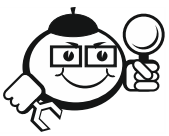
Например, для приведенной выше грамматики $N = \{S, X, Y\}$, $\Sigma = \{a, b, d, p, q, x, y\}$.

4.1 LL(k)-ГРАММАТИКИ

Если возможно написать детерминированный анализатор, осуществляющий разбор *сверху вниз*, для языка, генерируемого s-грамматикой, то такой анализатор принято называть **LL(1)-*грамматикой***. В общем случае, разбор сверху вниз можно осуществлять при помощи LL(k)-грамматики, где $k \geq 1$.

Обозначения в написании LL(k)-грамматики означают:

- L – строки разбираются слева направо;
- L – используются выводы из левых частей правил к правым;
- k – варианты порождающего правила выбираются с помощью предварительного просмотра k символов.


Пример

Добавим в рассмотренную выше грамматику еще одно правило:

$S \to pX$

$S \to qY$

$X \to aXb$

$X \to x$

$Y \to aYd$

$Y \to aS$

$Y \to y$

Далее попробуем проверить, принадлежит ли данному языку цепочка $qaapxd$. Начинаем с первого символа – q, следовательно, используем для вывода правило $S \to qY$:

$$S \Rightarrow^1 qY.$$

Переходим к следующему символу – a. И здесь уже однозначный вывод о выборе порождающего правила сделать нельзя, т.к. можно использовать как правило $Y \to aYd$, так и правило $Y \to aS$. Следовательно, нужно посмотреть на символ, следующий за a. Если это будет a или y, выбираем правило $Y \to aYd$ ($S \Rightarrow^2 qaYd$), а если это будет p или q – правило $Y \to aS$ ($S \Rightarrow^2 qaS$). То есть при разборе нам требуется заглядывать на 2 символа вперед, а грамматика является грамматикой типа LL(2).

. . . . . .

На практике работа с грамматиками при $k > 1$ слишком сложна, поэтому далее будем рассматривать лишь LL(1)-грамматики. LL(1)-грамматика очень удобна для организации процесса семантического разбора. Из определения LL(1)-грамматики следует, что эти грамматики можно разбирать детерминированно сверху вниз.

4.2 LL(1)-грамматики

4.2.1 Алгоритм проверки грамматики

Прежде чем строить таблицу разбора, необходимо убедиться, что грамматика является LL(1)-грамматикой [2].

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.

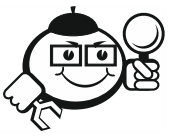
Если требуются дополнительные просмотры (т.е. значения некоторых элементов массива все еще имеют значение *UNDECIDED*), выполняются следующие действия.

1) Каждое порождающее правило, имеющее такой символ в правой части, который не может генерировать пустую цепочку (о чем свидетельствует значение соответствующего элемента массива), исклю-

чается из грамматики. В том случае, когда для нетерминала в левой части исключенного правила не существует других порождающих правил, значение элемента массива, соответствующего этому нетерминалу, устанавливается на *NO*.

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

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



Рассмотрим этот процесс на примере грамматики:

$A \to XYZ$

$X \to PQ$

$Y \to RS$

$R \to TU$

$P \to e$

$P \to a$

$Q \to aa$

$Q \to e$

$S \to c$

$T \to dd$

$U \to ff$

$Z \to e$

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

$A \to XYZ$

$X \to PQ$

$Y \to RS$

$R \to TU$

Таблица 4.2 – Массив после первого прохода

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

Здесь *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:

$A \to XYZ$

Таблица 4.3 – Массив после второго прохода

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). Все правила из грамматики исключены.

Таблица 4.4 – Массив после третьего прохода

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:

$P \to QR$

$Q \to qR$

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 – Пример матрицы предшественников

	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:

$$P \to QR$$

$$Q \to qR$$

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)-й и (j, k)-й позициях 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, l)-ю позицию, и повторять его до тех пор, пока не будет таких случаев, когда в (i, j)-й и в (j, k)-й позициях 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 – Матрица предшественников

	A	B	C	…	X	Y	Z	a	…	z
A		1			1			1		
B					1			1		
C										
…										
X								1		
Y										
Z										

. . . . . .

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).

4.2.2 Алгоритм поиска направляющих символов

Для построения таблицы разбора LL(1)-грамматики требуется определить множество направляющих символов для всех правил грамматики. Кроме того, эти множества также позволяют проверить, является ли грамматика корректной LL(1)-грамматикой.

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.

Для нахождения множества направляющих символов предварительно необходимо найти элементы двух вспомогательных множеств – множества предшествующих символов и множества последующих символов.

4.2.2.1 Множество предшествующих символов

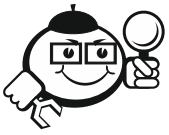
Множество терминальных ***символов-предшественников*** (от англ. start) определяется следующим образом:

$$a \in S(\alpha) \Leftrightarrow \alpha \Rightarrow^{*} a\beta,$$

где:

- a – терминал или пустая цепочка, $a \in \Sigma \cup \{e\}$;
- α и β – произвольные цепочки терминалов и/или нетерминалов, $\alpha, \beta \in (N \cup \Sigma)^{*}$;
- $S(\alpha)$ – множество символов-предшественников цепочки α.

То есть во множество символов-предшественников входят такие терминалы, которые могут появиться в начале цепочки, выводимой из α. Пустая цепочка также может являться элементом множества $S(\alpha)$, если она выводится из α.



Дана грамматика:

$$P \to Ac$$

$$P \to Bd$$

$$A \to a$$

$$A \to Aa$$

$$B \to b$$

$$B \to bB$$

Здесь символы a и b – символы-предшественники для P.

Чтобы найти предшествующие символы, можно построить дерево вывода для цепочки α. Причем, т.к. нас интересуют лишь терминалы, появляющиеся в начале цепочек, следует:

1) использовать самые левые выводы (т.е. самый левый нетерминал цепочки заменять правыми частями соответствующих порождающих правил);
2) при появлении в дереве вывода цепочки, начинающейся с терминала, терминал включать в искомое множество, а дальнейшее поддерево не строить, даже если в цепочке еще остались нетерминалы;
3) при появлении в дереве вывода пустой цепочки ее также включать в искомое множество.

Например, рассмотрим следующую грамматику:

$A \to BCDF$

$B \to 1 \mid 2 \mid \text{e}$

$C \to a \mid b \mid \text{e}$

$D \to x$

$F \to y$

Найдем множество символов-предшественников для α = *A*. Дерево вывода, построенное по описанным выше правилам, приведено на рис. 4.1. Стрелкой «↑» показаны позиции, в которых должны выводиться искомые терминалы. Жирным выделены найденные символы-предшественники.

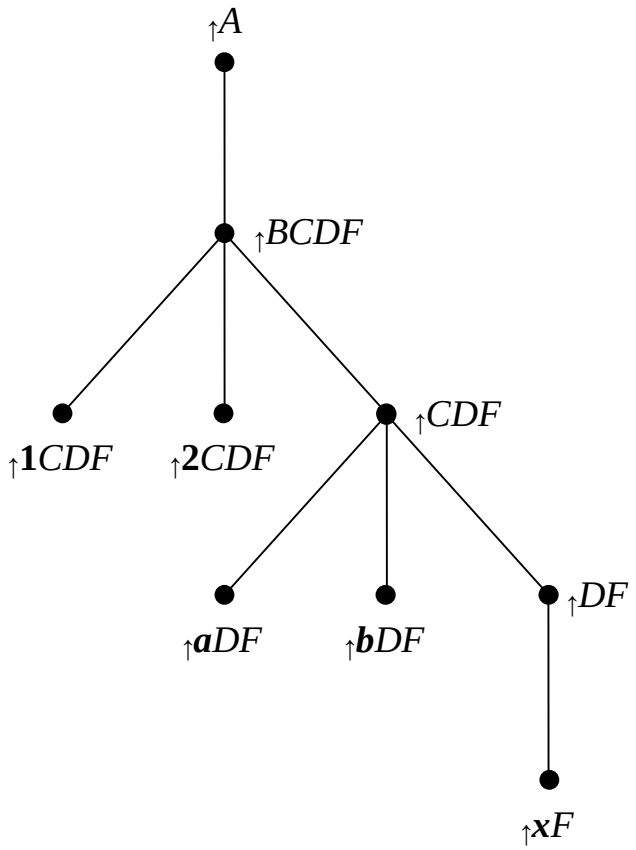


Рисунок 4.1 – Дерево вывода для $S(A)$

Итак,

$A \Rightarrow^2 \mathbf{1}CDF$

$A \Rightarrow^2 \mathbf{2}CDF$

$A \Rightarrow^3 \boldsymbol{a}DF$

$A \Rightarrow^3 \boldsymbol{b}DF$

$A \Rightarrow^4 \boldsymbol{x}F$

Следовательно, $S(A) = \{1, 2, a, b, x\}$. Аналогично можно строить множество предшествующих символов для любых других цепочек, составленных из элементов грамматики. Например:

$S(BD) = \{1, 2, x\}$,

$S(FDB) = \{y\}$,

$S(BBF) = \{1, 2, y\}$,

$S(BxB) = \{1, 2, x\}$,

$S(BC) = \{1, 2, a, b, e\}$,

$S(bAC) = \{b\}$.

В этом несложно убедиться, построив деревья выводов для этих цепочек.

. . . . . .

Можно также сделать некоторые частные выводы:

– $S(a\beta) = \{a\}$, где $a \in \Sigma$, а β – произвольная цепочка;

– $S(e) = \{e\}$.

Обобщая сказанное, получим формальное выражение для нахождения элементов множества предшествующих символов. Пусть $\alpha = X_1X_2\ldots X_n$, где $X_i \in N \cup \Sigma \cup \{e\}$. Тогда:

$$S(\alpha) = \bigcup_{i=1}^{k} \left(S(X_i) - \{e\} \right) \cup \Delta,$$

где:

$$k = \begin{cases} j \,\middle|\, e \notin S(X_j) \wedge e \in S(X_i), i < j, \\ n \,\middle|\, e \in S(X_i), i = 1, 2, \ldots, n, \end{cases}$$

$$\Delta = \begin{cases} \{e\} \,\middle|\, e \in S(X_i), i = 1, 2, \ldots, n, \\ \varnothing \,\middle|\, e \notin S(X_j) \wedge e \in S(X_i), i \neq j. \end{cases}$$

То есть k – это номер первого символа цепочки, для которого $e \notin S(X_k)$. Если же пустая цепочка присутствует во всех $S(X_i)$, $i = 1,2,\ldots,n$, то $k = n$. Пустая цепочка войдет во множество $S(\alpha)$ только в том случае, если для любого X_i, $i = 1,2,\ldots,n$, выполняется условие $e \in S(X_i)$. В этом случае получаем $\Delta = \{e\}$, в противном случае $\Delta = \varnothing$.

Для нахождения множества символов-предшественников для левых частей всех правил грамматики $G = (N, \Sigma, P, S)$ существует следующий нерекурсивный алгоритм:

1. Для всех правил грамматики $(A \to \alpha) \in P$ положить $S(A) = \varnothing$.
2. Для каждого правила грамматики $(A \to \alpha) \in P$ добавить к множеству $S(A)$ элементы множества $S(\alpha)$, полученные по приведенной выше формуле:

$$S(A) = S(A) \cup S(\alpha).$$

   При этом

$$S(X_i) = \begin{cases} \{X_i\} \mid X_i \in \Sigma \cup \{e\}, \\ \bigcup_j S(X_j) \mid X_i \in N \wedge (X_j \to \beta) \in P \wedge X_j = X_i. \end{cases}$$

   То есть для терминала или пустой цепочки во множестве $S(X_i)$ будет лишь один элемент – данный терминал или пустая цепочка. Для нетерминала множество $S(X_i)$ получается путем объединения всех множеств символов-предшественников тех правил грамматики, у которых данный нетерминал стоит слева от знака вывода.
3. Если при выполнении шага 2 хотя бы в одном множестве $S(A)$ появился хотя бы один новый элемент, вернуться на шаг 2. Иначе алгоритм заканчивает свою работу.

Пример

Рассмотрим работу этого алгоритма на следующем примере:

$$E \to T\ E'$$

$$E' \to +\ T\ E' \mid e$$

$$T \to F\ T'$$

$$T' \to *\ F\ T' \mid e$$

$$F \to (\ E\ ) \mid a$$

или, что то же самое:

$$E \to T\ E'$$

$E' \to + \; T \; E'$

$E' \to e$

$T \to F \; T'$

$T' \to * \; F \; T'$

$T' \to e$

$F \to (\; E \;)$

$F \to a$

Это уже рассмотренная ранее грамматика для математического выражения, содержащего операции сложения и умножения, а также скобки и операнды a, преобразованная к виду LL(1). Имеем грамматику

$$G = (N, \Sigma, P, S) = (\{E, E', T, T', F\}, \{+, *, (,), a\}, P, E).$$

В табл. 4.7 приведены результаты последовательных проходов второго шага алгоритма.

Таблица 4.7 – Результаты проходов алгоритма поиска символов-предшественников

P \ S	1-й проход	2-й проход	3-й проход
$E \to T \; E'$			(, a
$E' \to + \; T \; E'$	+	+	+
$E' \to e$	e	e	e
$T \to F \; T'$		(, a	(, a
$T' \to * \; F \; T'$	*	*	*
$T' \to e$	e	e	e
$F \to (\; E \;)$	(	(	(
$F \to a$	a	a	a

При четвертом проходе новые элементы во множествах $S(A)$ не появятся, поэтому он будет последним.

. . . . . .

4.2.2.2 Множество последующих символов

Множество терминальных ***последующих символов*** (от англ. follow) определяется следующим образом:

$$a \in F(A) \Leftrightarrow \alpha A \beta \Rightarrow^{*} \alpha A a \gamma,$$

где:

- a – терминал или признак конца цепочки, $a \in \Sigma \cup \{\perp\}$;
- α, β и γ – произвольные цепочки терминалов и/или нетерминалов, α, β, $\gamma \in (N \cup \Sigma)^{*}$;
- A – нетерминал, $A \in N$;
- $F(A)$ – множество последующих символов для нетерминала A.

Множество последующих символов включает в себя символы, которые могут следовать в выводимых цепочках за нетерминалом A. При этом данное множество не может содержать пустую цепочку. Но если при выводе после нетерминала A входная цепочка может заканчиваться, то множество $F(A)$ будет содержать специальный символ – маркер конца цепочки. Это может быть любой символ, не входящий в алфавит языка Σ, а также не совпадающий по написанию с нетерминалами грамматики N и символом пустой цепочки e.

Для нахождения множества последующих символов некоторого нетерминала A также строятся деревья вывода. При этом:

1) деревья строятся для правой части порождающего правила грамматики, содержащего нетерминал A, т.е. для правил вида $B \to \alpha A \beta$;
2) используются не самые левые выводы, как это было при нахождении множества символов-предшественников, а вывод из первого элемента цепочки β;
3) если после нереминала A в цепочке появился терминал, дальнейшее поддерево строить не следует, а терминал включается в искомое множество;

4) если после нетерминала A цепочка β может заканчиваться ($\beta \Rightarrow^* e$), тогда дальнейшее поддерево строится для всех цепочек правых частей правил грамматики, содержащих нетерминал B, причем в них B заменяется полученной ранее при выводе цепочкой;
5) если после нетерминала A цепочка β может заканчиваться ($\beta \Rightarrow^* e$) и $A = S$ (т.е. A – это стартовый нетерминал), то в искомое множество включается маркер конца цепочки.

Например, найдем для грамматики:

$A \to BCD$

$B \to aCx \mid bCd \mid F$

$C \to zBD \mid c \mid e$

$D \to Fx \mid e$

$F \to y$

символы, следующие за нетерминалом B (рис. 4.2). Стрелкой «↑» показаны позиции, в которых должны выводиться искомые терминалы. Жирным выделены найденные последующие символы.

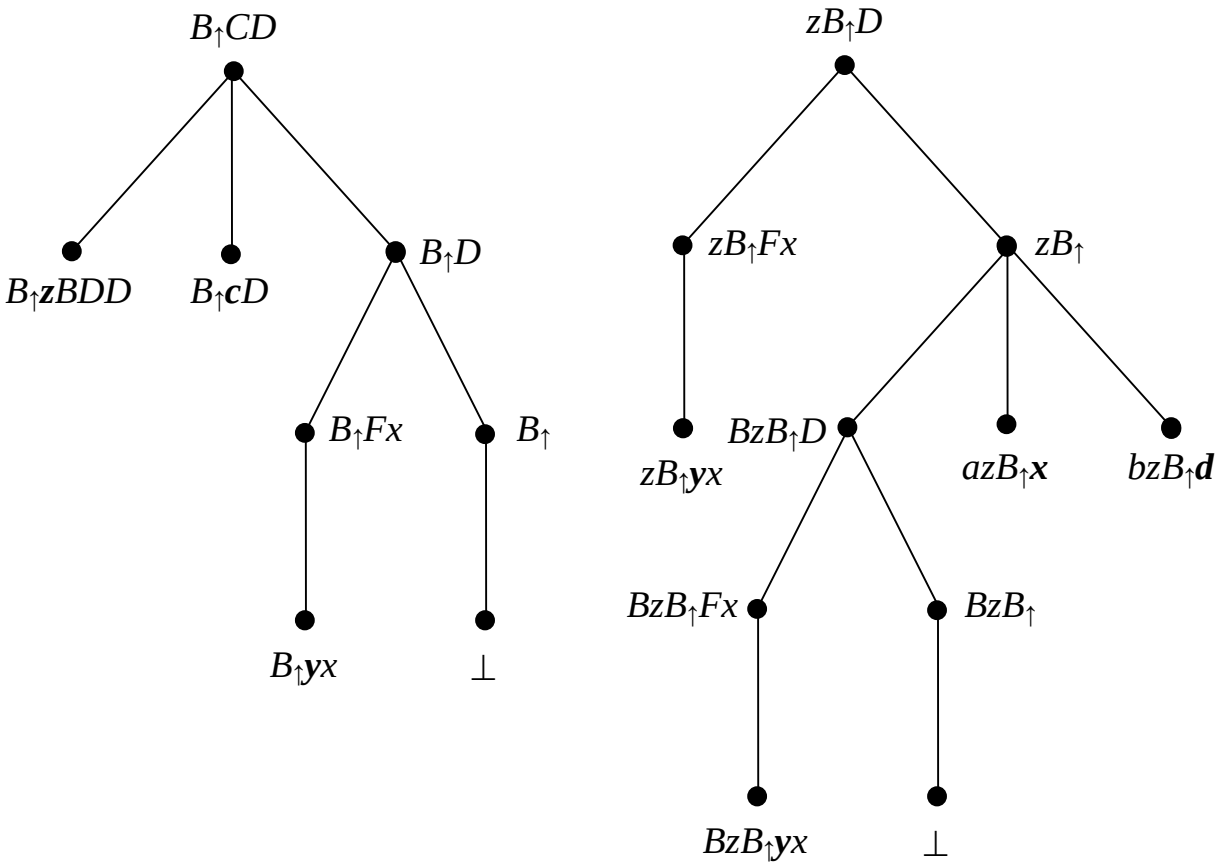


Рисунок 4.2 – Деревья вывода для $F(B)$ и $F(zB)$

Для первого дерева, используя пункты 1–3, получаем:

$$B_{\uparrow}CD \Rightarrow^{1} B_{\uparrow}\mathbf{z}BDD$$

$$B_{\uparrow}CD \Rightarrow^{1} B_{\uparrow}\mathbf{c}D$$

$$B_{\uparrow}CD \Rightarrow^{3} B_{\Uparrow}\mathbf{y}x$$

Но $BCD \Rightarrow^{2} B$ (или $CD \Rightarrow^{2} e$), т.е. нетерминал B оказывается в конце цепочки. В левой части правила $A \to BCD$ находится нетерминал A, но в правых частях правил он не встречается, поэтому пункт 4 неприменим. Так как при этом A является стартовым символом, то добавляем к множеству $F(B)$ маркер конца цепочки согласно пункту 5. Получили $F(B) = \{c, y, z, \perp\}$.

Во втором дереве по пунктам 1–3 получаем вывод

$$zB_{\uparrow}D \Rightarrow^{2} zB_{\Uparrow}\mathbf{y}x$$

Далее, т.к. $zB_{\uparrow}D \Rightarrow^{1} zB_{\uparrow}$ (или $D \Rightarrow^{1} e$) и в левой части правила $C \to zBD$ находится нетерминал C, находим все правые части порождающих правил, содержащих данный нетерминал. Таких правил три:

$$A \to BCD$$

$$B \to aCx$$

$$B \to bCd$$

Используя пункт 4, подставляем в них вместо C цепочку zB и продолжаем процесс. Таким образом, для первого вхождения C получаем:

$$BCD \Rightarrow^{*} BzB_{\uparrow}D \Rightarrow BzB_{\uparrow}Fx \Rightarrow BzB_{\uparrow}\boldsymbol{y}x$$

$$BCD \Rightarrow^{*} BzB_{\uparrow}D \Rightarrow BzB_{\uparrow}$$

Снова, как и выше, получается ситуация, когда во множество $F(B)$ следует добавить маркер конца цепочки (в левой части правила $A \to BCD$ находится стартовый символ). Для второго и третьего вхождения C в грамматику:

$$aCx \Rightarrow^{*} azB_{\uparrow}\boldsymbol{x}$$

$$bCd \Rightarrow^{*} bzB_{\uparrow}\boldsymbol{d}$$

В результате, учитывая полученные ранее элементы, $F(B) = \{d, x, y, \perp\} \cup \{c, y, z, \perp\} = \{c, d, x, y, z, \perp\}$.

. . . . . .

По сути, для правила грамматики вида $B \to \alpha A \beta$ во множество $F(A)$ входят элементы множества $S(\beta)$, кроме e, а если после нетерминала A цепочка β может заканчиваться ($\beta \Rightarrow^{*} e$), то также и элементы множества $F(B)$:

$$F(A) = \left(S(\beta) - \{e\}\right) \cup \begin{cases} F(B) \mid \beta = e \vee e \in S(\beta), \\ \varnothing \mid \beta \neq e \wedge e \notin S(\beta). \end{cases}$$

Для нахождения множества последующих символов для нетерминалов грамматики $G = (N, \Sigma, P, S)$ существует следующий нерекурсивный алгоритм:

1. Для всех нетерминалов грамматики $A \in N$ положить $F(A) = \varnothing$. Для стартового нетерминала положить $F(S) = \{\perp\}$.

2. Для каждого вхождения нетерминала A в правую часть порождающих правил грамматики вида $(B \to \alpha A \beta) \in P$ добавить к множеству $F(A)$ новые элементы, найденные по приведенной выше формуле:

$$F(A) = F(A) \cup \left(S(\beta) - \{e\}\right) \cup \begin{cases} F(B) \mid \beta = e \vee e \in S(\beta), \\ \varnothing \mid \beta \neq e \wedge e \notin S(\beta). \end{cases}$$

3. Если при выполнении шага 2 хотя бы в одном множестве $F(A)$ появился хотя бы один новый элемент, вернуться на шаг 2. Иначе алгоритм заканчивает свою работу.

Пример

Рассмотрим работу этого алгоритма на примере грамматики:

$E \to T\ E'$

$E' \to +\ T\ E' \mid e$

$T \to F\ T'$

$T' \to *\ F\ T' \mid e$

$F \to (\ E\ ) \mid a$

В табл. 4.8 приведены результаты последовательных проходов второго шага алгоритма.

Таблица 4.8 – Результаты проходов алгоритма поиска последующих символов

N \ F	1-й проход	2-й проход
E	⊥,)	⊥,)
E'	⊥,	⊥,)
T	⊥, +	⊥, +,)
T'	⊥, +	⊥, +,)
F	⊥, +, *	⊥, +, *,)

При третьем проходе новые элементы во множествах $F(B)$ не появятся, поэтому он будет последним.

. . . . . .

4.2.2.3 Множество направляющих символов

Если A – нетерминал в левой части правила, то его ***направляющими символами*** $T(A)$ будут символы-предшественники A и все символы, следующие за A, если A может генерировать пустую строку:

$$T(A) = \left(S(A) - \{e\}\right) \cup \begin{cases} F(A) \mid e \in S(A), \\ \varnothing \mid e \notin S(A). \end{cases}$$

. . . . .  Пример . . . . . .

Для рассмотренной выше грамматики получим результат, приведенный в табл. 4.9.

Таблица 4.9 – Множества направляющих символов

P	$S(A)$	$F(A)$	$T(A)$
$E \to T\ E'$	(, a	⊥,)	(, a
$E' \to +\ T\ E'$	+	⊥,)	+
$E' \to e$	e		⊥,)
$T \to F\ T'$	(, a	⊥, +,)	(, a
$T' \to *\ F\ T'$	*	⊥, +,)	*
$T' \to e$	e		⊥, +,)
$F \to (\ E\ )$	(	⊥, +, *,)	(
$F \to a$	a		a

. . . . . .

Обратите внимание, что множество символов-предшественников ищется для цепочки, а множество предшествующих символов – для нетерминала.

Поэтому для альтернатив порождающего правила с одним и тем же нетерминалом A в левой части множества $S(A)$ будут разными, а множества $F(A)$ будут совпадать.

Как следует из данного ранее определения, для грамматики типа LL(1) множества направляющих символов для порождающих правил с одинаковым символом в левой части не должны пересекаться. То есть, если имеются альтернативы порождающего правила:

$$A_1 \to \alpha_1$$

$$A_2 \to \alpha_2$$

$$\ldots\ldots\ldots$$

$$A_n \to \alpha_n$$

и $A_1 = A_2 = \ldots = A_n$ соответственно, то

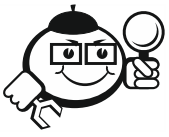
$$T(A_i) \cap T(A_j) = \varnothing \text{ при } i \neq j.$$

В рассмотренной выше грамматике это условие выполняется. Из этого условия следуют и другие требования к LL(1)-грамматикам. В частности, запрет левой рекурсии. Действительно, если в грамматике есть правила вида:

$$A \to A\alpha$$

$$A \to \beta$$

то элементы множества $S(\beta)$ войдут во множества $T(A)$ для обеих альтернатив.



Пример

Рассмотрим грамматику:

$$E \to T + T$$

$$E \to T * T$$

$$E \to T$$

$$T \to (E)$$

$T \to a$

Это не LL(1)-грамматика, т.к. для всех трех альтернатив порождающих правил с символом E в левой части получим $T(E) = \{(, a\}$.

. . . . . .

Для нетерминала A в правой части правила $B \to \alpha A \beta$ его направляющие символы определяются так:

$$T(A) = \begin{cases} \left(S(A\beta) - \{e\}\right) \cup F(B) \,\middle|\, e \in S(A\beta), \\ S(A\beta) \,\middle|\, e \notin S(A\beta). \end{cases}$$

4.3 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 LL(1)-грамматики в общем виде представляет собой одномерный массив структур следующего вида [2]:

```
declare 1 TABLE,
          2 terminals LIST,
          2 jump int,
          2 accept bool,
          2 stack bool,
          2 return bool,
```

```
                2 error bool;
```

где

```
     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 → *begin* DECLIST *semi* STATLIST *end*

(2) DECLIST → *d* X

(3) X → *comma* DECLIST

(4) X → *e*

(5) STATLIST → *s* Y

(6) Y → *comma* STATLIST

(7) Y → *e*

Сначала представим грамматику в виде схемы (рис. 4.3). В скобках и справа на рисунке указаны номера элементов таблицы разбора.

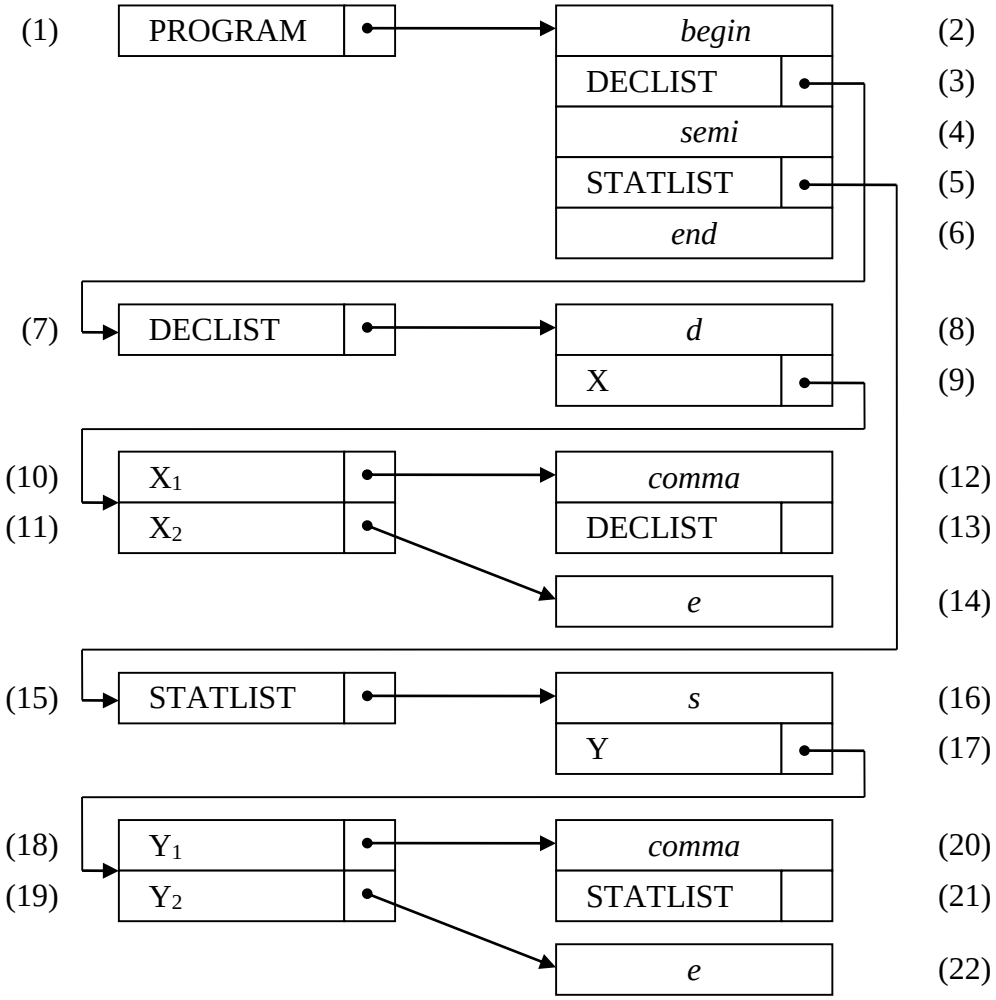


Рисунок 4.3 – Схема грамматики

Таблица разбора, соответствующая этой грамматике, может быть представлена в виде табл. 4.10.

Таблица 4.10 – 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

. . . . . .

4.3.1 Построение таблицы

Алгоритм построения таблицы разбора следующий:

1. Разметить грамматику. При этом порядковые номера $i \in M$ присваиваются всем элементам грамматики, от первого правила к последнему, от левого символа к правому. Как видно из рис. 4.3, при наличии у порождающего правила альтернатив сначала нумеруются левые части всех альтернативных правил, а уже затем их правые части. Еще одним важным требованием является то, что все альтернативные правила должны следовать друг за другом в списке правил.

2. Построить два вспомогательных множества $M_L \subset M$ и $M_R \subset M$. В первое включить порядковые номера $i \in M$ элементов грамматики, расположен-

ных в левой части порождающего правила, во второе – порядковые номера элементов, которыми заканчиваются правые части порождающих правил:

$$M_L = \{i \mid (X_i \to \alpha) \in P\},$$

$$M_R = \{j \mid (A \to \alpha X_j) \in P\}.$$

3. Построить таблицу, состоящую из столбцов **terminals**, **jump**, **accept**, **stack**, **return**, **error**. Количество строк таблицы определяется количеством элементов во множестве M – по одной строке на каждый элемент грамматики.

4. Заполнить ячейки таблицы.

4.1. Для нетерминала множество **terminals** совпадает с множеством направляющих символов. Для терминала множество **terminals** включает лишь сам терминал. Для пустой цепочки множество **terminals** совпадает с множеством направляющих символов соответствующего правила, у которого данная пустая цепочка стоит в правой части:

$$\mathbf{terminals}_i = \begin{cases} T(X_i) \mid X_i \in N, \\ \{X_i\} \mid X_i \in \Sigma, \\ T(X_j) \mid X_i = e \wedge (X_j \to X_i) \in P. \end{cases}$$

4.2. Переход **jump** от нетерминала в левой части правила осуществляется к первому символу правой части этого правила. Переход от нетерминала в правой части правила осуществляется на такой же нетерминал в левой части. Причем если имеется несколько альтернатив соответствующего порождающего правила, переход осуществляется к первой из альтернатив. От терминала, не последнего в цепочке правой части правила, переход осуществляется к следующему символу цепочки. Если терминал завершает цепочку либо это символ e (он в правой части может быть только один, поэтому будет одновременно начинать и завершать ее), то значение **jump** равно нулю:

$$\mathbf{jump}_i = \begin{cases} k \,\big|\, i \in M_L \wedge \left(X_i \to X_k \alpha\right) \in P, \\ k \,\big|\, X_i \in N \wedge i \notin M_L \wedge X_k = X_i \wedge j \in M_L \wedge j-1 \notin M_L, \\ i+1 \,\big|\, i \in \Sigma \wedge i \notin M_R, \\ 0 \,\big|\, i \in \Sigma \cup \{e\} \wedge i \in M_R. \end{cases}$$

4.3. Символ принимается (**accept**), если это терминал:

$$\mathbf{accept}_i = \begin{cases} \text{true} \,\big|\, X_i \in \Sigma, \\ \text{false} \,\big|\, X_i \notin \Sigma. \end{cases}$$

4.4. Номер строки таблицы разбора помещается в стек (**stack**), если соответствующий символ грамматики – нетерминал в правой части порождающего правила, но не в конце цепочки правой части:

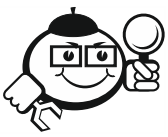
$$\mathbf{stack}_i = \begin{cases} \text{true} \,\big|\, X_i \in N \wedge i \notin M_L \wedge i \notin M_R, \\ \text{false} \,\big|\, X_i \notin N \vee i \in M_L \vee i \in M_R. \end{cases}$$

4.5. Возврат (**return**) по стеку при разборе осуществляется, если символ грамматики является терминалом, расположенным в конце цепочки правой части порождающего правила, или символом e (т.е. когда $\mathbf{jump}_i = 0$):

$$\mathbf{return}_i = \begin{cases} \text{true} \,\big|\, i \in \Sigma \cup \{e\} \wedge i \in M_R, \\ \text{false} \,\big|\, i \notin \Sigma \cup \{e\} \vee i \notin M_R. \end{cases}$$

4.6. Ошибка (**error**) при разборе не генерируется, если следующий символ грамматики находится в левой части альтернативного порождающего правила:

$$\mathbf{error}_i = \begin{cases} \text{false} \,\big|\, i \in M_L \wedge i+1 \in M_L, \\ \text{true} \,\big|\, i \notin M_L \vee i+1 \notin M_L. \end{cases}$$



Пример

Маркировка грамматики для языка математических выражений будет иметь следующий вид:

$$E_1 \to T_2\ E'_3$$

$$E'_4 \to +_6\ T_7\ E'_8$$

$$E'_5 \to e_9$$

$$T_{10} \to F_{11}\ T'_{12}$$

$$T'_{13} \to *_{15}\ F_{16}\ T'_{17}$$

$$T'_{14} \to e_{18}$$

$$F_{19} \to (_{21}\ E_{22}\ )_{23}$$

$$F_{20} \to a_{24}$$

Здесь уже не допускается использование символа «|» для записи альтернативных порождающих правил, т.к. их левые части будут маркироваться разными метками. В грамматике 24 элемента X_i, $i \in M$, $M = \{1, 2, \ldots, 24\}$, следовательно, в таблице разбора будет 24 строки. При этом:

$$M_L = \{1, 4, 5, 10, 13, 14, 19, 20\},$$

$$M_R = \{3, 8, 9, 12, 17, 18, 23, 24\}.$$

Строим таблицу разбора (табл. 4.11).

Таблица 4.11 – Таблица разбора

i	*X*	**terminals**	**jump**	**accept**	**stack**	**return**	**error**
1	*E*	(, *a*	2				
2	*T*	(, *a*	10		true		
3	*E′*	+,), ⊥	4				
4	*E′*	+	6				false
5	*E′*	), ⊥	9				
6	+	+	7	true			
7	*T*	(, *a*	10		true		
8	*E′*	+,), ⊥	4				

9	*e*	), ⊥	0			true	
10	*T*	(, *a*	11				
11	*F*	(, *a*	19		true		
12	*T′*	+, *,), ⊥	13				
13	*T′*	*	15				false
14	*T′*	+,), ⊥	18				
15	*	*	16	true			
16	*F*	(, *a*	19		true		
17	*T′*	+, *,), ⊥	13				
18	*e*	+,), ⊥	0			true	
19	*F*	(	21				false
20	*F*	*a*	24				
21	(	(	22	true			
22	*E*	(, *a*	1		true		
23	)	)	0	true		true	
24	*a*	*a*	0	true		true	

Все остальные ячейки **accept**, **stack** и **return** содержат false, а все остальные ячейки **error** – true.

. . . . . .

4.3.2 Разбор цепочки по таблице

Для разбора цепочки $\alpha = a_1 a_2 \ldots a_n \perp$ нам потребуется магазин (стек) *M*. Операцию помещения элемента в стек будем обозначать как $M \leftarrow x$, операцию извлечения элемента из стека – как $M \rightarrow x$. Номер текущей строки таблицы разбора обозначим как *i*, номер текущего символа во входной строке – *k*.

Алгоритм разбора цепочки по таблице:

1. Положить *i* := 1 (разбор начинается с первой строки таблицы).

2. Положить $k := 1$ (разбор идет слева направо, начиная с первого символа цепочки).
3. $M \leftarrow 0$ (поместить в стек значение 0).
4. Если $a_k \in \mathbf{terminals}_i$, то:
   4.1. Если $\mathbf{accept}_i$ = true, то $k := k + 1$ (перейти к следующему символу входной цепочки).
   4.2. Если $\mathbf{stack}_i$ = true, то $M \leftarrow i$ (поместить в стек номер текущей строки таблицы разбора).
   4.3. Если $\mathbf{return}_i$ = true, то:
      4.3.1. $M \rightarrow i$;
      4.3.2. Если $i = 0$, то перейти на шаг 6;
      4.3.3. $i := i + 1$;
      4.3.4. Вернуться на шаг 4.
   4.4. Если $\mathbf{jump}_i \neq 0$, то:
      4.4.1. $i := \mathbf{jump}_i$;
      4.4.2. Вернуться на шаг 4.
5. Иначе если $\mathbf{error}_i$ = false, то у правила есть еще одна альтернатива и нужно просто перейти к следующей строке таблицы разбора:
   5.1. $i := i + 1$;
   5.2. Вернуться на шаг 4.
6. В противном случае разбор окончен. Если при этом стек M пуст, а $a_k = \perp$, то разбор завершен успешно. Иначе цепочка содержит синтаксическую ошибку и k – позиция этой ошибки.

Пример

Рассмотрим разбор предложения

begin d comma d semi s semi end $\perp$

по табл. 4.10. Действия приведены в табл. 4.12.

Таблица 4.12 – Разбор предложения

№	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; 3 помещается в стек; перейти к 7	3 0
7	*d* принимается; перейти к 8	3 0
8	*d* проверяется и принимается; перейти к 9	3 0
9	*comma* считывается и проверяется; перейти к 10	3 0
10	*comma* проверяется; перейти к 12	3 0
12	*comma* проверяется и принимается; перейти к 13	3 0
13	*d* считывается и проверяется; перейти к 7	3 0
7	*d* проверяется; перейти к 8	3 0
8	*d* проверяется и принимается; перейти к 9	3 0
9	*comma* считывается и проверяется; перейти к 10	3 0
10	*semi* не совпадает с *comma*; ошибка – «ложь», перейти к 11	3 0
11	*comma* проверяется; перейти к 14	3

		0
14	*semi* проверяется; возврат – «истина», удаляется 3; перейти к 4	0
4	*semi* проверяется и принимается; перейти к 5	0
5	*s* считывается и проверяется; 5 помещается в стек; перейти к 15	5 0
15	*s* проверяется; перейти к 16	5 0
16	*s* проверяется и принимается; перейти к 17	5 0
17	*comma* считывается и проверяется; перейти к 18	5 0
18	*comma* проверяется; перейти к 20	5 0
20	*comma* проверяется и принимается; перейти к 21	5 0
21	*s* считывается и проверяется; перейти к 15	5 0
15	*s* проверяется; перейти к 16	5 0
16	*s* проверяется и принимается; перейти к 17	5 0
17	*end* считывается и проверяется; перейти к 18	5 0
18	*end* не совпадает с *comma*; ошибка – «ложь», перейти к 19	5 0
19	*end* проверяется; перейти к 22	5

		0
22	*end* проверяется; возврат – «истина», удалить 5; перейти к 6	0
6	*end* проверяется и принимается; возврат – «истина», удалить 0; разбор заканчивается	

Метод разбора LL(1) имеет ряд преимуществ:

1) Никогда не требует возврата, поскольку этот метод детерминирован.
2) Время разбора пропорционально длине программы.
3) Имеются хорошие диагностические характеристики, 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 по главе 4

1. Эквивалентность МП-автоматов и КС-грамматик.
2. LL(*k*)-грамматики.
3. *s*-грамматика.
4. LL(1)-грамматика.
5. Алгоритм определения принадлежности данной грамматики к LL(1)-грамматике.
6. LL(1)-таблица разбора.
7. Разбор по таблице.

5 СИНТАКСИЧЕСКИЙ АНАЛИЗ СНИЗУ ВВЕРХ

5.1 LR(K)-ГРАММАТИКИ

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* [3, 4, 5].

Обозначения в написании LR(k)-грамматики означают:

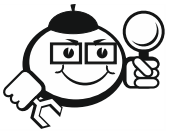
– L – строки разбираются слева направо;

– R – используются свертки правых частей правил к левым;

– k – выбор между сдвигом и сверткой производится с помощью предварительного просмотра k символов.

Дадим определение понятиям сдвига и свертки. Синтаксический анализатор, работающий по принципу «снизу вверх», выполняет действия двух типов [2]:

1) *сдвиг* – считывание и помещение символа в стек, что соответствует движению на один пункт вдоль какого-нибудь правила грамматики;
2) *свертка* (*приведение*) – замещение множества элементов в верхней части стека каким-либо нетерминалом грамматики с помощью одного из порождающих правил этой грамматики.



Пример

Рассмотрим, например, язык, генерируемый правилами:

(1) *VARS* → **real** *ID IDLIST*

(2) *IDLIST* → , *ID IDLIST*

(3) *IDLIST* → ;

(4) *ID* → *a-z*

Здесь *a-z* – это диапазон символов от *a* до *z*. Правила специально подобраны так, чтобы удовлетворять как LL-, так и LR-грамматике. В чем будут различия при разборе предложения «**real** *a*, *b*, *c*;» методом сверху вниз и методом снизу вверх?

Разбор сверху вниз (LL) начинается со стартового символа *S* = *VARS*, а его цель – спускаясь вниз, найти в дереве вывода лист «**real** *a*, *b*, *c*;». Если такой лист будет найден, то предложение принадлежит языку, иначе нет:

$$\begin{aligned} VARS &\Rightarrow^{1} \textbf{real}\ ID\ IDLIST \\ &\Rightarrow^{2} \textbf{real}\ a\ IDLIST \\ &\Rightarrow^{3} \textbf{real}\ a, ID\ IDLIST \\ &\Rightarrow^{4} \textbf{real}\ a, b\ IDLIST \\ &\Rightarrow^{5} \textbf{real}\ a, b, ID\ IDLIST \\ &\Rightarrow^{6} \textbf{real}\ a, b, c\ IDLIST \\ &\Rightarrow^{7} \textbf{real}\ a, b, c; \end{aligned}$$

. . . . . .

Сентенциальная форма *– это последовательность символов (терминалов и нетерминалов), выводимых из начального символа грамматики.*

. . . . . .

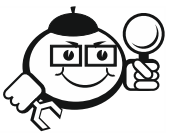
Соответственно, если

$$S \Rightarrow_l^* \upsilon$$

в процессе левостороннего вывода, то υ – *левая сентенциальная форма*.

По аналогии с левой сентенциальной формой, последовательность символов ω, выводимых из начального символа грамматики в процессе правостороннего вывода, называется *правой сентенциальной формой*:

$$S \Rightarrow_r^* \omega$$



Разбор снизу вверх (LR) идет в обратном порядке – предполагая, что предложение «**real** *a, b, c*;» – лист дерева вывода, делается попытка подняться вверх к корню дерева (стартовому символу *S* = *VARS*). Если это удается, то данная цепочка принадлежит языку. Таким образом, синтаксические анализаторы, работающие по принципу LR, сводят предложения языка к начальному символу путем последовательного применения правил грамматики.

Для LR-разбора используется стек. Каждый символ, считанный анализатором, немедленно помещается в стек анализатора:

real↑ *a, b, c*;

real

real *a*↑, *b, c*;

a
real

Это действие называется «*сдвиг*» (англ. shift). Стрелка ставится непосредственно после последнего считанного символа. Затем анализатор заменяет *a* с помощью правила (4) на нетерминал в его левой части. Это действие называется «*приведение*» или «*свертка*» (англ. reduce). Правые части правил заменяются их соответствующими левыми частями:

real *a*↑, *b, c*;

ID
real

Далее анализатор считывает следующие символы:

real *a*,↑ *b*, *c*;

,
ID
real

real *a*, *b*↑, *c*;

b
,
ID
real

Снова применяется свертка по правилу (4):

real *a*, *b*↑, *c*;

ID
,
ID
real

Считываем два следующих символа:

real *a*, *b*,↑ *c*;

,
ID
,
ID
real

real *a*, *b*, *c*↑;

c
,
ID
,
ID
real

Очередное действия – приведение с использованием правила (4):

real *a*, *b*, *c*↑;

ID

,
ID
,
ID
real

Считываем последний символ:

real *a*, *b*, *c*;↑	;
	ID
	,
	ID
	,
	ID
	real

Выполняем приведение по правилу (3):

real *a*, *b*, *c*↑	*IDLIST*
	ID
	,
	ID
	,
	ID
	real

Теперь три верхних символа на стеке можно свернуть по правилу (2):

real *a*, *b*, *c*↑	*IDLIST*
	ID
	,
	ID
	real

Затем еще раз:

real *a*, *b*, *c*↑ | *IDLIST* / *ID* / **real**

И последнее действие – свертка по правилу (1):

real *a*, *b*, *c*↑ | *VARS*

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

В итоге получили следующий процесс разбора:

real *a*, *b*, *c*; $\Rightarrow^1$ **real** *ID*, *b*, *c*;

$\Rightarrow^2$ **real** *ID*, *ID*, *c*;

$\Rightarrow^3$ **real** *ID*, *ID*, *ID*;

$\Rightarrow^4$ **real** *ID*, *ID*, *ID IDLIST*

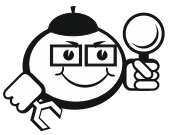
$\Rightarrow^5$ **real** *ID*, *ID IDLIST*

$\Rightarrow^6$ **real** *ID IDLIST*

$\Rightarrow^7$ *VARS*

. . . . . .

Однако данная грамматика весьма проста, к тому же праволинейна. В ней всегда легко выбрать альтернативу порождающего правила при LL-разборе либо сделать выбор между сверткой и сдвигом при LR-разборе. Но так бывает далеко не всегда.

. . . . .  Пример . . . . . .

Например, рассмотрим следующую LL-грамматику:

$E \to F\ E'$

$E' \to ADD$

$E' \to MUL$

$E' \to e$

$ADD \to + E$

$MUL \to * E$

$F \to (E)$

$F \to a$

Попробуем вывести из нее цепочку «(*a* + *a*)**a*»:

$E \Rightarrow^1 F\ E'$

$\Rightarrow^2 (E)\ E'$

$\Rightarrow^3 (F\ E')\ E'$

$\Rightarrow^4 (a\ E')\ E'$

А далее возникают затруднения. У порождающего правила E' есть три альтернативы, но узнать, какую из них выбрать, не заглядывая вперед, невозможно. Если же выбрать неверную альтернативу, то придется возвращаться назад, что противоречит принципам LL-разбора. Для этого и строится LL-таблица – по ней выбор альтернативы осуществляется однозначно.

Подобные проблемы возникают и при LR-разборе. Для примера рассмотрим грамматику:

(1) *VARS* → **real** *IDLIST* ;

(2) *IDLIST* → *ID* , *IDLIST*

(3) *IDLIST* → *ID*

(4) *ID* → *a-z*

Возьмем предложение «**real** *a, b, c*;» и начнем его приведение к стартовому символу:

real *a, b, c*; $\Rightarrow^1$ **real** *ID, b, c*;

$\Rightarrow^2$ **real** *IDLIST, b, c*;

$\Rightarrow^3$ **real** *IDLIST, ID, c*;

$\Rightarrow^4$ **real** *IDLIST, IDLIST, c*;

$\Rightarrow^5$ **real** *IDLIST*, *IDLIST*, *ID*;

$\Rightarrow^6$ **real** *IDLIST*, *IDLIST*, *IDLIST*;

Несмотря на то, что предложение является верным, разбор зашел в тупик. Причина в том, что на втором шаге была выполнена лишняя процедура свертки. Необходимо было сделать сдвиг:

real *a*, *b*, *c*; $\Rightarrow^1$ **real** *ID* , *b*, *c*;

$\Rightarrow^2$ **real** *ID* , *ID* , *c*;

$\Rightarrow^3$ **real** *ID* , *ID* , *ID* ;

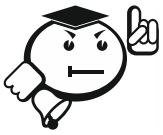
$\Rightarrow^4$ **real** *ID* , *ID* , *IDLIST* ;

$\Rightarrow^5$ **real** *ID* , *IDLIST* ;

$\Rightarrow^6$ **real** *IDLIST* ;

$\Rightarrow^7$ *VARS*

Как и в предыдущем примере, заранее выбрать, что предпочтительнее – сдвиг или свертка – в общем случае невозможно, а возвраты противоречат принципам LR-разбора. Следовательно, для успешного разбора требуется построение LR-таблицы.



Учитывая, что свертку правила вида $A \to e$ (*e*-правила) можно проводить сколь угодно раз в любой ситуации, очевидно, что грамматика типа LR не может иметь *e*-правил. С другой стороны, при использовании правостороннего вывода отсутствуют ограничения LL-грамматик – множества направляющих символов в LR-грамматиках могут пересекаться, и, в частности, разрешена левая рекурсия.

5.2 LR(1)-ГРАММАТИКИ

На практике обычно используют LR(k)-таблицы разбора для k = 0 или k = 1. При LR(0)-разборе для выбора дальнейших действий входная цепочка вообще не используется – поведение анализатора зависит лишь от содержимого стека. При LR(1)-разборе принимается во внимание также один текущий символ из входной цепочки. Грамматики для $k > 1$ на практике не используются. Далее будем рассматривать лишь LR(1)-разбор.

Введем некоторые определения для LR(1)-анализаторов (далее, говоря LR, будем подразумевать LR(1)).

. . . . . .

Конфигурацией *LR-анализатора* (*LR-конфигурацией*) *будем называть пару вида*:

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

где S_i *– состояния LR-анализатора* ($S_i \in \mathcal{I}$, $\mathcal{I}$ *– множество всех состояний*), X_i *– элементы грамматики* ($X_i \in \mathcal{X}$, $\mathcal{X}$ *– множество всех элементов грамматики,* $\mathcal{X} = N \cup \Sigma$), *а* a_i *– терминалы из входной цепочки* ($a_i \in \Sigma$).

. . . . . .

При этом последовательность элементов $S_0\ X_1\ S_1\ X_2\ S_2 \ldots\ X_m\ S_m$ представляет собой содержимое магазина (стека) анализатора, а символы $a_i a_{i+1} \ldots a_n$ – непрочитанную часть входной цепочки.

Согласно данному ранее определению, правая сентенциальная форма может быть получена конкатенацией элементов грамматики, расположенных на стеке, с непрочитанной частью входной цепочки:

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

Префикс данной цепочки $X_1 X_2 \ldots X_m$ называется *активным префиксом* LR-анализатора.

Текущее состояние анализатора находится на вершине стека – это S_m. Текущий символ входной цепочки – a_i.

Рассмотрим, как меняется конфигурация анализатора при сдвиге и свертке:

1) Если в состоянии S_m для текущего символа входной цепочки a_i таблицей разбора T приписывается сдвиг в состояние S_k, т.е. $T(S_m, a_i) = S_k$, то на стек помещается новая пара $(a_i\ S_k)$, а во входной цепочке происходит переход к следующему символу:

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

То есть новым состоянием анализатора будет S_k, а текущим символом входной цепочки станет a_{i+1}.

2) Если в состоянии S_m для текущего символа входной цепочки a_i таблицей приписывается свертка по правилу P_k, т.е. $T(S_m, a_i) = R_k$, то:

2.1) Со стека снимается r пар символов, где r – количество элементов в правой части правила $P_k = (A \to \alpha) \in P$. При этом на вершине стека остается состояние S_{m-r}:

$$(S_0\ X_1\ S_1\ X_2\ S_2 \ldots\ X_{m-r}\ S_{m-r},\ a_i a_{i+1} \ldots a_n \perp),\ r = |\alpha|.$$

2.2) На стек помещается пара $(A\ S_j)$, где A – левая часть правила P_k, а $S_j = T(S_{m-r}, A)$:

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

Входная цепочка при этом остается без изменений. Новым состоянием анализатора становится S_j.

5.3 LR(1)-ТАБЛИЦА РАЗБОРА

Выше мы рассмотрели технологию разбора «снизу вверх». Однако в некоторых ситуациях нужно решать, следует ли выполнять конкретное приведение, когда правая часть правила появляется в вершине стека, или выпол-

нять сдвиг. Механизмом, который регулирует вопрос принятия правильного решения, является таблица разбора.

Имеется несколько, незначительно отличающихся, вариантов построения таблицы разбора. В том варианте, который рассматривается в пособии, на грамматику накладывается дополнительное ограничение – стартовый символ не должен иметь альтернатив порождающего правила. Чтобы это гарантировать, в грамматику вводится новый искусственный стартовый символ S' и новое искусственное стартовое правило $S' \to S$. Таким образом, получаем модифицированную грамматику:

$$G' = (N', \Sigma, P', S'),$$

где $N' = N \cup \{S'\}$, $P' = \{S' \to S\} \cup P$.

Таблица T для грамматики $G' = (N', \Sigma, P', S')$ представляет собой матрицу (табл. 5.1). Ее столбцы можно условно разделить на две секции. Одна из них (секция действий, англ. action) состоит из столбцов для каждого терминала грамматики a_1, a_2, …, $a_n \in \Sigma$ и маркера конца цепочки «⊥». Вторая (секция переходов, англ. goto) состоит из столбцов для нетерминалов грамматики A_1, A_2, …, $A_m \in N$. В принципе, это разделение не является обязательным. Строки таблицы соответствуют каждому состоянию S_0, S_1, …, в котором может находиться анализатор. Каждое состояние соответствует той позиции в порождающем правиле, которой достиг анализатор. Заранее количество состояний неизвестно. Их будет столько, сколько удастся найти уникальных состояний анализатора при построении множества состояний $\mathcal{J}$.

Таблица 5.1 – Общий вид LR(1)-таблицы

$\mathcal{X}$ / $\mathcal{J}$	*Action*					*Goto*			
	a_1	a_2	…	a_n	⊥	A_1	A_2	…	A_m
S_0									
S_1									
…									

Драйвер анализатора использует два стека – *стек символов* и *стек состояний*, хотя, как было показано выше, символы и состояния можно хранить в одном стеке, чередуя их. Секция действий таблицы разбора включает элементы четырех типов:

1) *Элементы сдвига* $T(S_m, a_i) = S_k$. Это означает, что в стек символов помещается символ a_i, а в стек состояний – состояние S_k. Либо, если стек один, в него помещается пара (a_i S_k).
2) *Элемент приведения* $T(S_m, a_i) = R_k$. Это означает, что необходимо выполнить свертку с помощью P_k – правила с номером k из грамматики (т.е. предварительно все правила должны быть пронумерованы). Если в правой части правила P_k находится r символов, то удаляется r элементов из стека символов и r элементов из стека состояний (либо r пар элементов из общего стека). Затем на стек символов помещается нетерминал A из левой части правила P_k, а на стек состояний – состояние S_j, определенное по описанным выше правилам.
3) *Элемент ошибок* $T(S_m, a_i) = ERROR$. Эти элементы являются пробелами в таблице и соответствуют синтаксическим ошибкам.
4) *Элемент остановки* $T(S_m, a_i) = HALT$. Означает успешное завершение разбора.

Секция переходов используется лишь для определения нового состояния S_j при свертке.

Заметим, что состояния S_i – это элементы множества состояний $\boldsymbol{\mathcal{I}}$. Алгоритм построения $\boldsymbol{\mathcal{I}}$, который будет рассмотрен далее, таков, что лишь начальное состояние S_0 всегда будет получено первым, а порядок расположения остальных состояний может быть любым. На практике это означает, что строки таблицы могут меняться местами (с соответствующим изменением нумераций сдвигов). То есть одной и той же LL-грамматике всегда соответ-

ствует лишь одна таблица разбора, а для LR-грамматики таблицы могут различаться. Но этот факт на конечный результат разбора влияния не оказывает.

Пример

Например, рассмотрим грамматику:

VARS → **real** *IDLIST* ;

IDLIST → *IDLIST* , *ID*

IDLIST → *ID*

ID → *a-z*

Модифицированная грамматика (с новым стартовым символом *VARS′* и новым стартовым правилом) будет иметь следующий вид:

(0) *VARS′* → *VARS*

(1) *VARS* → **real** *IDLIST* ;

(2) *IDLIST* → *IDLIST* , *ID*

(3) *IDLIST* → *ID*

(4) *ID* → *a-z*

А таблица разбора может иметь вид табл. 5.2.

Таблица 5.2 – Пример таблицы разбора

$\mathcal{I}$	**real**	*a-z*	,	;	⊥	*VARS*	*IDLIST*	*ID*
S_0	S_2					S_1		
S_1					*HALT*			
S_2		S_5					S_3	S_4
S_3			S_7	S_6				
S_4			R_3	R_3				
S_5			R_4	R_4				
S_6					R_1			
S_7		S_5						S_8

S_8			R_2	R_2				

5.3.1 Состояния анализатора

Решим задачу нахождения множества состояний LR-анализатора.

Состояние анализатора, в свою очередь, является множеством, элементами которого являются *LR-ситуации*. Таким образом, множество состояний LR-анализатора – это множество множеств (мультимножество) LR-ситуаций.

LR-ситуация *– это пара вида*:

$$[A \to \alpha \bullet \beta \mid a],$$

где $(A \to \alpha\beta) \in P$ *– правило грамматики,* $a \in \Sigma \cup \{\perp\}$ *– один из множества символов, которые могут появиться на входе анализатора* (*это могут быть терминальные символы или маркер окончания входной цепочки*).

То есть LR-ситуация – это сопоставление правила грамматики, правая часть которого разделена точкой на две части, и некоторого терминального символа. Если в правой части грамматики n элементов, то возможен n+1 вариант разделения.

Пример

Например, для правила *IDLIST* → *IDLIST* , *ID* это будут:

$[IDLIST \to \bullet\ IDLIST\ ,\ ID \mid a]$,

$[IDLIST \to IDLIST \bullet\ ,\ ID \mid a]$,

$[IDLIST \to IDLIST\ , \bullet\ ID \mid a]$,

$[IDLIST \to IDLIST\ ,\ ID \bullet \mid a]$.

Точка указывает на ту позицию, до которой правая часть правила уже распознана. Терминальный символ указывает на то, что, когда правая часть будет распознана полностью (последняя ситуация), возможно выполнение приведения или свертки при наличии во входной цепочке символа *a* (*символ приведения* или *символ свертки*).

Пример

Например:

[*IDLIST* → *IDLIST* , • *ID* | ;]

Это означает, что правая часть правила распознана по запятую включительно, а когда она будет полностью распознана и во входной строке встретится терминал «;», будет выполнена свертка по правилу *IDLIST* → *IDLIST* , *ID*.

Граф переходов *между состояниями LR-анализатора – это помеченный граф* $\mathcal{G} = (\mathcal{I}, R)$*, вершинами которого являются состояния* $I \in \mathcal{I}$*, а разметка* g *отображает множество дуг* R ($R \subset \mathcal{I} \times \mathcal{I}$) *во множество* $\mathcal{X}$.

Алгоритм построения множества состояний и графа переходов LR-анализатора:

1. Формируем состояние, состоящее лишь из одной LR-ситуации [$S' \to$ • $S \mid \perp$], т.е. за ее основу берется стартовое правило ($S' \to S$)$\in P'$, точка ставится в начале правой части, а в качестве символа приведения берется маркер конца входной цепочки. Далее выполняем замыка-

ние этого состояния (алгоритм работы этой функции рассмотрим ниже), что даст начальное состояние анализатора S_0:

$$S_0 = \textit{ЗАМЫКАНИЕ}(\{[S' \to \bullet\, S \mid \perp]\}).$$

2. Полагаем, что множество состояний LR-анализатора $\mathcal{J}$ пока состоит только из одного состояния – S_0:

   $$\mathcal{J} = \{S_0\},$$

   а граф переходов не имеет дуг:

   $$R = \varnothing.$$

3. Организуем два вложенных цикла:

   – для каждого состояния $I \in \mathcal{J}$,

   – для каждого символа грамматики $X \in \mathcal{X}$.

   3.1. Строим новое состояние, вызывая функцию перехода (ее алгоритм также дан ниже) для аргументов I и X:

   $$J = \textit{ПЕРЕХОД}(I, X).$$

   3.2. Если $J \neq \varnothing$ и $J \notin \mathcal{J}$, т.е. состояние J не пустое уникальное (во множестве состояний нет точно такого же состояния), добавляем его во множество состояний:

   $$\mathcal{J} = \mathcal{J} \cup \{J\}.$$

   3.3. Если $J \neq \varnothing$ и $(I, J) \notin R$, т.е. состояние J не пустое и в графе переходов $\mathcal{G}$ отсутствует дуга, ведущая от вершины I в вершину J, добавляем ее в граф и помечаем меткой X:

   $$R = R \cup (I, J),\ g((I, J)) = X.$$

4. Если в процессе выполнения шага 3 во множество состояний $\mathcal{J}$ было добавлено хотя бы одно новое состояние или был добавлен хотя бы один новый переход в граф $\mathcal{G}$, вернуться на шаг 3. Иначе построение множества $\mathcal{J}$ закончено.

Алгоритм функции замыкания *ЗАМЫКАНИЕ*(I):

Вход: Аргумент I – некоторое LR-состояние.

Выход: Измененное состояние I с новыми LR-ситуациями. Цель замыкания – найти все ситуации в грамматике, в которых состояние анализатора (следовательно, и дальнейшее поведение) идентично.

1. Организуем три вложенных цикла:

   – для каждой LR-ситуации вида $[A \to \alpha \bullet B\beta \mid a] \in I$, $A \in N'$, $B \in N$, $\alpha \in \mathcal{X}^*$, $\beta \in \mathcal{X}^*$, $a \in \Sigma \cup \{\perp\}$, т.е. ситуаций, где точка стоит перед нетерминалом,

   – для каждого терминала b из множества символов-предшественников цепочки βa, т.е. $b \in S(\beta a)$, $b \in \Sigma \cup \{\perp\}$,

   – для каждого правила вывода $(B \to \gamma) \in P$, т.е. правила, у которого в левой части находится нетерминал B (перед которым стояла точка в рассматриваемой LR-ситуации).

   1.1. Формируем новую LR-ситуацию вида:

   $$\sigma = [B \to \bullet \gamma \mid b].$$

   1.2. Если $\sigma \notin I$, т.е. в состоянии I не было ситуации σ, добавляем ее:

   $$I = I \cup \{\sigma\}.$$

2. Если в процессе выполнения шага 1 в состояние I была добавлена хотя бы одна новая ситуация, вернуться на шаг 1. Иначе работа функции закончена.

Для LR-грамматик нахождение множества символов-предшественников осуществляется гораздо проще, чем для LL-грамматик, т.к. в них отсутствуют e-правила. Если вспомнить формулу нахождения множества символов-предшественников цепочки $\alpha = X_1 X_2 \ldots X_n$:

$$S(\alpha) = \bigcup_{i=1}^{k} \left(S(X_i) - \{e\} \right) \cup \Delta,$$

и учесть, что в LR-грамматиках не используются e-правила, т.е. $e \notin S(X_i)$, то получим:

$$S(\alpha) = S(X_1).$$

Алгоритм функции перехода *ПЕРЕХОД*(*I*, *X*):

Вход: Аргументы I – некоторое LR-состояние, $X \in \mathfrak{X}$ – символ грамматики.

Выход: Новое состояние J, полученное после распознания символа X в правых частях правил грамматики (т.е. перенос точки в LR-ситуациях через символ X).

1. Полагаем $J = \varnothing$.
2. Для всех ситуаций $[A \to \alpha \bullet X\beta \mid a] \in I$, т.е. ситуаций исходного состояния, в которых точка стоит перед X, добавляем в состояние J такую же ситуацию, но с точкой после X:
$$J = J \cup \{[A \to \alpha X \bullet \beta \mid a]\}.$$
3. Результатом будет *ЗАМЫКАНИЕ*(*J*).

Пример

Применение описанного алгоритма к грамматике:

(0) $VARS' \to VARS$

(1) $VARS \to$ **real** $IDLIST$;

(2) $IDLIST \to IDLIST$, ID

(3) $IDLIST \to ID$

(4) $ID \to a\text{-}z$

даст следующее множество состояний:

$S_0 = \{ [VARS' \to \bullet\ VARS \mid \perp],$

$[VARS \to \bullet$ **real** $IDLIST$; $\mid \perp]\};$

$S_1 = \{ [VARS' \to VARS \bullet \mid \perp]\};$

$S_2 = \{ [VARS \to$ **real** $\bullet\ IDLIST$; $\mid \perp],$

$[IDLIST \to \bullet\ IDLIST$, $ID \mid$;],

[*IDLIST* → • *ID* | ;],

[*IDLIST* → • *IDLIST* , *ID* | ,],

[*IDLIST* → • *ID* | ,],

[*ID* → • *a-z* | ;],

[*ID* → • *a-z* | ,]};

S_3 = { [*VARS* → **real** *IDLIST* • ; | ⊥],

[*IDLIST* → *IDLIST* • , *ID* | ;],

[*IDLIST* → *IDLIST* • , *ID* | ,]};

S_4 = { [*IDLIST* → *ID* • | ;],

[*IDLIST* → *ID* • | ,]};

S_5 = { [*ID* → *a-z* • | ;],

[*ID* → *a-z* • | ,]};

S_6 = { [*VARS* → **real** *IDLIST* ; • | ⊥]};

S_7 = { [*IDLIST* → *IDLIST* , • *ID* | ;],

[*IDLIST* → *IDLIST* , • *ID* | ,],

[*ID* → • *a-z* | ;],

[*ID* → • *a-z* | ,]};

S_8 = { [*IDLIST* → *IDLIST* , *ID* • | ;],

[*IDLIST* → *IDLIST* , *ID* • | ,]};

Получили $\mathcal{I}$ = {S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8}. Дуги графа переходов $\mathcal{G}$ = ($\mathcal{I}$, R) и его разметка g будут следующими:

$$R = \{(S_0, S_1), (S_0, S_2), (S_2, S_3), (S_2, S_4), (S_2, S_5), (S_3, S_6), (S_3, S_7), (S_7, S_8), (S_7, S_5)\},$$

$g(S_0, S_1)$ = *VARS*, $g(S_0, S_2)$ = **real**, $g(S_2, S_3)$ = *IDLIST*, $g(S_2, S_4)$ = *ID*, $g(S_2, S_5)$ = *a-z*, $g(S_3, S_6)$ = «;», $g(S_3, S_7)$ = «,», $g(S_7, S_8)$ = *ID*, $g(S_7, S_5)$ = *a-z*.

Сам граф $\mathcal{G}$ изображен на рис. 5.1.

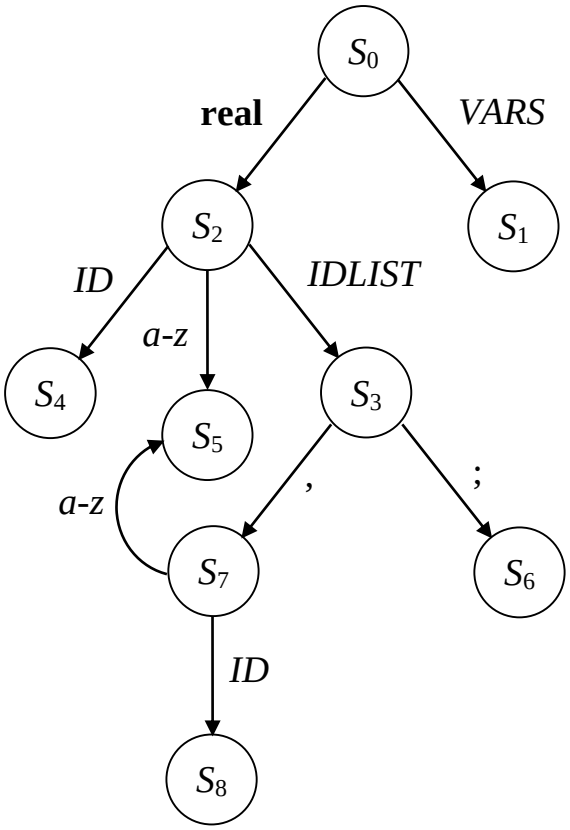


Рисунок 5.1 – Граф переходов между состояниями анализатора

. . . . . .

5.3.2 Построение таблицы

Алгоритм построения LR-таблицы разбора следующий:

1. Создаем таблицу из $\#\mathcal{X}+1$ столбца (по столбцу на каждый терминал $a \in \Sigma$ и нетерминал $A \in N$, а также столбец для маркера конца цепочки «⊥») и $\#\mathcal{I}$ строк (по одной на каждое состояние анализатора).
2. В каждую ячейку таблицы записываем элемент *ERROR*.
3. Для всех состояний $S_i \in \mathcal{I}$ выполняем следующие действия:
   3.1. Если:

   а) в состоянии S_i присутствует LR-ситуация с точкой не в конце правила, т.е. $[A \to \alpha \bullet X\beta \mid a] \in S_i$, $A \in N'$, $X \in \mathcal{X}$, $\alpha \in \mathcal{X}^*$, $\beta \in \mathcal{X}^*$, $a \in \Sigma \cup \{\perp\}$ и

б) в графе переходов есть переход из состояния S_i в некоторое состояние S_j, помеченный меткой X, т.е. $(S_i, S_j) \in R$, $g((S_i, S_j)) = X$, то $T(S_i, X) = S_j$.

3.2. Если в состоянии S_i присутствует LR-ситуация с точкой в конце правила, т.е. $[A \to \alpha \bullet \mid a] \in S_i$, $A \in N'$, $\alpha \in \mathfrak{X}^+$, $a \in \Sigma \cup \{\perp\}$, то:

а) если $(A \to \alpha) \in P'$ – это искусственное стартовое правило, т.е. $A = S'$ (в этом случае $a = \perp$), то $T(S_i, a) = HALT$,

б) иначе $T(S_i, a) = R_k$, где k – номер правила $(A \to \alpha) \in P$ в грамматике, т.е. $P_k = (A \to \alpha)$.

Пример

Если применить данный алгоритм к грамматике

(0) $VARS' \to VARS$

(1) $VARS \to$ **real** $IDLIST$;

(2) $IDLIST \to IDLIST$, ID

(3) $IDLIST \to ID$

(4) $ID \to a\text{-}z$

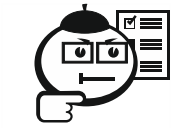
то получим табл. 5.2.

5.3.3 LR-конфликты

Если таблица для грамматики типа LL заполняется построчно, то в описанном выше алгоритме элементы заносятся в грамматику точечно. Может сложиться такая ситуация, когда при попытке записи в ячейку элемента сдвига или свертки ячейка уже содержит другой элемент сдвига или свертки. Такая ситуация называется ***LR-конфликтом***. В общем случае конфликты могут быть четырех типов:

1. При попытке записать в ячейку сдвиг S_i, обнаруживаем, что там уже содержится свертка R_j (конфликт типа сдвиг-свертка);
2. При попытке записать в ячейку свертку R_i, обнаруживаем, что там уже содержится сдвиг S_j (конфликт типа свертка-сдвиг);
3. При попытке записать в ячейку сдвиг S_i, обнаруживаем, что там уже содержится другой сдвиг S_j (конфликт типа сдвиг-сдвиг);
4. При попытке записать в ячейку свертку R_i обнаруживаем, что там уже содержится другая свертка R_j (конфликт типа свертка-свертка).

Последние два случая на практике встречаются редко, а первые два встречаются весьма часто. Например, возникновение конфликтов типа сдвиг-сдвиг или свертка-свертка возможно при наличии в грамматике двух и более одинаковых правил.



Другой пример (проверьте самостоятельно):

$S \to A\ S$

$S \to S\ A$

$S \to A\ S\ A$

$S \to \text{a}$

$A \to \text{a}$

$A \to A\ A$

В подобных случаях делается вывод о том, что грамматика некорректна и дальнейшее построение таблицы разбора прекращается. При возникновении конфликта типа сдвиг-свертка или свертка-сдвиг необходимо *разрешить конфликт*, т.е. принять решение – записать новый элемент поверх старого или проигнорировать его. Как показали исследования, для большинства грамматик предпочтительным является сдвиг (это, в частности, демонстрирует последний пример в разделе 5.1). Поэтому при возникновении конфлик-

та типа сдвиг-свертка пишем в ячейку таблицы сдвиг S_i вместо свертки R_j. Если же возникает конфликт свертка-сдвиг, то оставляем в ячейке сдвиг S_j, а свертку R_i игнорируем. Хотя есть специфические грамматики, где свертка является более предпочтительной, они редко встречаются на практике.

5.3.4 Разбор цепочки по таблице

Для разбора, как уже было сказано, требуется один или два стека (магазина). В начале в стек состояний помещается состояние S_0 (или просто его индекс – 0). То есть конфигурацией LR-автомата будет

$$(S_0, a_1a_2\ldots a_n\perp).$$

Далее конфигурация меняется по описанным ранее правилам до тех пор, пока не будет достигнута ячейка таблицы с элементом *HALT* или *ERROR*. Во втором случае позицией ошибки будет позиция первого непрочитанного символа входной цепочки.

В табл. 5.3 рассмотрим, как с помощью табл. 5.2 разбирается строка «**real** *a*, *b*, *c*;⊥». В данном примере при разборе используются два стека.

Таблица 5.3 – Пример разбора строки

Позиция	Стек символов	Стек состояний
↑**real** *a*, *b*, *c*;⊥		0
Входной символ – **real**, $T(S_0$, **real**$) = S_2$, сдвиг в состояние 2:		
real↑ *a*, *b*, *c*;⊥	**real**	2
		0
Входной символ – *a*, $T(S_2$, *a*-*z*$) = S_5$, сдвиг в состояние 5:		
real *a*↑, *b*, *c*;⊥	*a*	5
	real	2

		0

Входной символ – «,», $T(S_5$, «,») = R_4, приведение по правилу 4 ($ID \rightarrow a\text{-}z$). В его левой части один элемент, снимаем со стеков по одному элементу:

real *a*↑, *b*, *c*;⊥	**real**	2
		0

На вершине стека остается состояние S_2. Следовательно, новым состоянием будет $T(S_2, ID) = S_4$:

real *a*↑, *b*, *c*;⊥	*ID*	4
	real	2
		0

Входной символ – «,», $T(S_4$, «,») = R_3, приведение по правилу 3 ($IDLIST \rightarrow ID$). Снимаем со стеков по одному элементу:

real *a*↑, *b*, *c*;⊥	**real**	2
		0

Новым состоянием будет $T(S_2, IDLIST) = S_3$:

real *a*↑, *b*, *c*;⊥	*IDLIST*	3
	real	2
		0

Входной символ – «,», $T(S_3$, «,») = S_7, сдвиг в состояние 7:

real *a*,↑ *b*, *c*;⊥	,	7
	IDLIST	3
	real	2
		0

Входной символ – *b*, $T(S_7, a\text{-}z) = S_5$, сдвиг в состояние 5:

real *a*, *b*↑, *c*;⊥	*b*	5
	,	7
	IDLIST	3
	real	2

		0

Входной символ – «,», $T(S_5, \text{«,»}) = R_4$, приведение по правилу 4 ($ID \rightarrow a\text{-}z$). Снимаем со стеков по одному элементу:

real *a*, *b*↑, *c*;⊥	,	7
	IDLIST	3
	real	2
		0

Новым состоянием будет $T(S_7, ID) = S_8$:

real *a*, *b*↑, *c*;⊥	*ID*	8
	,	7
	IDLIST	3
	real	2
		0

Входной символ – «,», $T(S_8, \text{«,»}) = R_2$, приведение по правилу 2 ($IDLIST \rightarrow IDLIST\ ,\ ID$). Снимаем со стеков по три элемента:

real *a*, *b*↑, *c*;⊥	**real**	2
		0

Новым состоянием будет $T(S_2, IDLIST) = S_3$:

real *a*, *b*↑, *c*;⊥	*IDLIST*	3
	real	2
		0

Входной символ – «,», $T(S_3, \text{«,»}) = S_7$, сдвиг в состояние 7:

real *a*, *b*,↑ *c*;⊥	,	7
	IDLIST	3
	real	2
		0

Входной символ *c* – $T(S_7, a\text{-}z) = S_5$, сдвиг в состояние 5:

real *a*, *b*, *c*↑;⊥	*c*	5

	,	7
	IDLIST	3
	real	2
		0

Входной символ – «;», $T(S_5, \text{«;»}) = R_4$, приведение по правилу 4 ($ID \rightarrow a\text{-}z$). Снимаем со стеков по одному элементу:

real *a*, *b*, *c*↑;⊥	,	7
	IDLIST	3
	real	2
		0

Новым состоянием будет $T(S_7, ID) = S_8$:

real *a*, *b*, *c*↑;⊥	*ID*	8
	,	7
	IDLIST	3
	real	2
		0

Входной символ – «;», $T(S_5, \text{«;»}) = R_2$, приведение по правилу 2 ($IDLIST \rightarrow IDLIST\ ,\ ID$). Снимаем со стеков по три элемента:

real *a*, *b*, *c*↑;⊥	**real**	2
		0

Новым состоянием будет $T(S_2, IDLIST) = S_3$:

real *a*, *b*, *c*↑;⊥	*IDLIST*	3
	real	2
		0

Входной символ – «;», $T(S_3, \text{«;»}) = S_6$, сдвиг в состояние 6:

real *a*, *b*, *c*; ↑⊥	;	6
	IDLIST	3
	real	2

	0

Входной символ – «⊥», $T(S_6, \perp) = R_1$, приведение по правилу 1 (*VARS* → **real** *IDLIST* ;). Снимаем со стеков по три элемента:

real *a*, *b*, *c*; ↑⊥		0

Новым состоянием будет $T(S_0, VARS) = S_1$:

VARS	1
	0

Входной символ – «⊥», $T(S_1, \perp) = HALT$, разбор завершен успешно.

. . . . . .

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 [2].

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 по главе 5

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 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.

6.1 Получение четверок

В качестве примера включения действия в грамматику для генерирования кода рассмотрим проблему разложения арифметических выражений на *четверки* [2]. Выражения определяются грамматикой со следующими правилами:

$S \to EXP$

$EXP \to TERM \mid EXP + TERM$

$TERM \to FACT \mid TERM \times FACT$

$FACT \to -FACT \mid ID \mid (EXP)$

$ID \to a \mid b \mid c \mid d \mid e$

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:

$QUAD \to OPER\ OP_1\ OPER = INT \mid OP_2\ OPER = INT$

$OPER \to INT \mid ID$

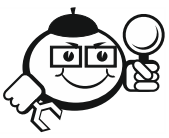
$INT \rightarrow DIGIT \mid DIGIT\ INT$

$DIGIT \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$ID \rightarrow a \mid b \mid c \mid d \mid e$

$OP_1 \rightarrow + \mid \times$

$OP_2 \rightarrow -$



Пример

Примеры четверок:

$$-a = 4$$

$$a + b = 7$$

$$6 + 3 = 11$$

Выражение

$$(-a + b)\times(c + d)$$

будет соответствовать следующей 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 A_1, A_2, … Для реализации алгоритма четверок требуется четыре действия. Алгоритм

пользуется стеком, а номера четверок размещаются с помощью целочисленной переменной. Перечислим эти действия:

– A_1 – поместить элемент в стек;
– A_2 – взять три элемента из стека, напечатать их с последующим знаком «=» и номером следующей размещаемой четверки и поместить полученное целое число в стек;
– A_3 – взять два элемента из стека, напечатать их с последующим значением «=» и номером следующей размещаемой четверки и поместить полученное целое в стек;
– A_4 – взять из стека один элемент.

Грамматика с учетом этих добавлений примет вид:

$$S \to EXP\ \langle A_4 \rangle$$

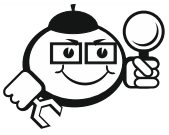
$$EXP \to TERM \mid EXP + \langle A_1 \rangle\ TERM\ \langle A_2 \rangle$$

$$TERM \to FACT \mid TERM \times \langle A_1 \rangle\ FACT\ \langle A_2 \rangle$$

$$FACT \to - \langle A_1 \rangle\ FACT \mid \langle A_3 \rangle\ ID\ \langle A_1 \rangle \mid (\ EXP\ )$$

$$ID \to a \mid b \mid c \mid d \mid e$$

Действие A_1 используется для помещения в стек всех идентификаторов и операторов, а действия A_2 и A_3 – для получения бинарных и унарных четверок соответственно.



Пример

В качестве примера проследим за преобразованием выражения

$$(-a + b)\times(c + d)$$

в четверки. Действие A_1 выполняется после распознавания каждого идентификатора и оператора, действие A_2 – после второго операнда каждого знака бинарной операции, а действие A_3 – после первого (и единственного) опера-

тора каждой унарной операции. Действие A_4 выполняется только один раз после считывания всего выражения (табл. 6.1).

Таблица 6.1 – Преобразование выражения в четверки

Последняя считанная литера	Действие	Выход
(	–	
–	A_1, поместить в стек «–»	
a	A_1, поместить в стек a	
	A_3, удалить из стека 2 элемента	$-a = 1$
	Поместить в стек «1»	
+	A_1, поместить в стек «+»	
b	A_1, поместить в стек b	
	A_2, удалить из стека 3 элемента	$1 + b = 2$
	Поместить в стек «2»	
)	–	
×	A_1, поместить в стек «×»	
(	–	
c	A_1, поместить в стек c	
+	A_1, поместить в стек «+»	
d	A_1, поместить в стек d	
	A_2, удалить из стека 3 элемента	$c + d = 3$
	Поместить в стек «3»	
)	–	
	A_2, удалить из стека 3 элемента	$2 \times 3 = 4$
	Поместить в стек «4»	
	A_4, удалить из стека 1 элемент	

. . . . . .

В рассматриваемом примере нам не пришлось сравнивать приоритеты двух операций, так как эти приоритеты уже заложены в правилах грамматики.

6.2 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 [2].

В принятой формализации описание

int *x*

является ***определяющей реализацией*** *x, а использование x в другом контексте*

x := 4 *или x* + *y или* **read**(y)

говорит, что имеется ***прикладная реализация*** *x.*

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.

Пример

Пусть синтаксис описания идентификаторов задается правилами:

DEC → **real** *IDS* | **integer** *IDS* | **boolean** *IDS*

IDS → *ID*

IDS → *IDS*, *ID*

а блок определяется как

BLOCK → **begin** *DECS*; *STATS* **end**,

где:

DECS → *DECS*; *DEC*

DECS → *DEC*

STATS → *STATS*; *stat*

STATS → *stat*

В этом случае структуру таблицы символов можно представить в виде рис. 6.1.

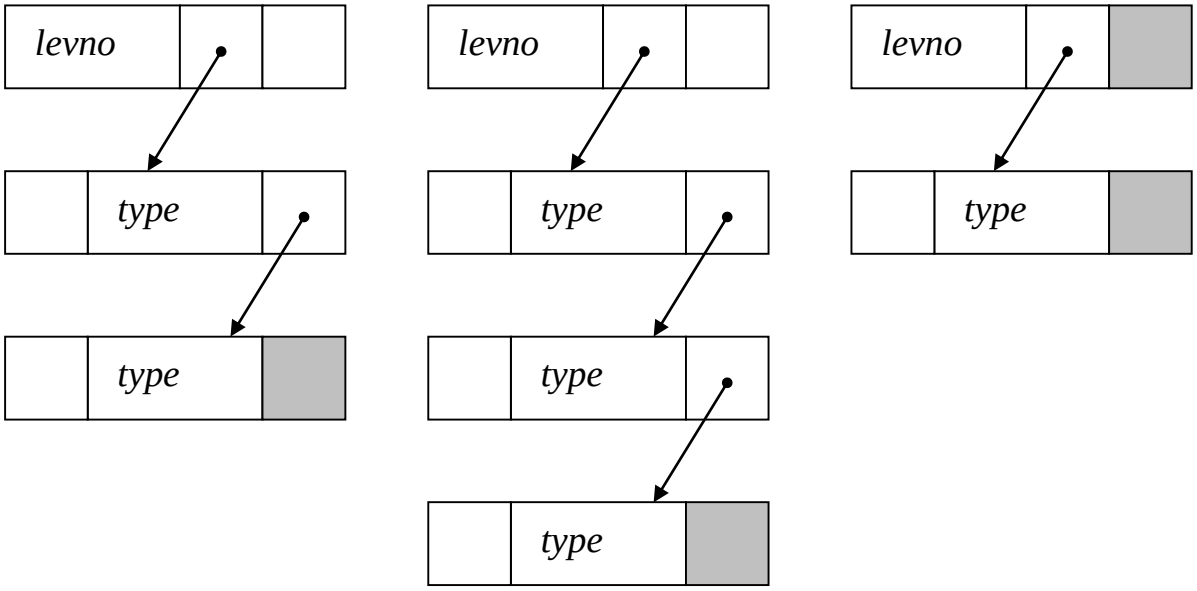


Рисунок 6.1 – Структура таблицы символов

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 свя-

занные с ними прикладные и определяющие реализации идентификаторов требуют поиска в стеке в направлении сверху вниз.

Пример

Рассмотренный метод иллюстрируется следующим примером (табл. 6.2).

Таблица 6.2 – Стек с элементами таблицы символов

Вид программы	Элемент таблицы	
	Имя	Тип
begin **int** *a*, *b*	*b* *a*	**int** **int**
…		
begin **char** *c*, *d*	*d* *c* *b* *a*	**char** **char** **int** **int**
…		
end	*b* *a*	**int** **int**
begin **int** *e*, *f*	*f* *e* *b* *a*	**int** **int** **int** **int**
begin **char** *g*	*g* *f* *e* *b*	**char** **int** **int** **int**

	a	**int**
…		
end	*f* *e* *b* *a*	**int** **int** **int** **int**
end	*b* *a*	**int** **int**
end		

Таким образом, включение действий в грамматику позволяет получить простой и элегантный компилятор. При этом действия выполняются на соответствующем уровне в грамматике.

Контрольные вопросы по главе 6

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.

7 Проектирование компиляторов

7.1 Число проходов

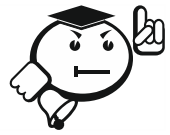
Разработчики компиляторов находят идею однопроходного компилятора привлекательной, так как не надо заботиться о связях между проходами, промежуточных языках и т.д. Кроме того, нет трудностей в ассоциировании ошибок программы с исходным тестом. Однако вопрос о количестве проходов неоднозначен [2]. Прежде всего, надо определить, что мы будем считать проходом.

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.

7.2 Таблицы символов

Информацию о типе (виде) идентификаторов синтаксический анализатор хранит с помощью таблицы символов (имен) [2]. Этими таблицами также пользуется 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 представлен 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.

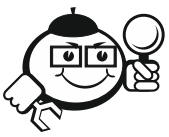
Если бы идентификаторы были расположены в алфавитном порядке, то поиск осуществлялся бы гораздо быстрее за счет последовательного деления таблицы пополам (*двоичный поиск*), однако на сортировку записей по порядку каждый раз при добавлении новой записи уходило бы очень много времени, что существенно 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 идентификато-

ров в программе, и определяется отображение каждого возможного идентификатора на элемент массива (функция отображения). 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 наличие 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.



Пример

После встречи с идентификаторами

CAR DOG CAB ASC EGG

таблица примет вид табл. 7.1; позиции идентификаторов зависят от порядка их внесения в таблицу.

Таблица 7.1 – Таблица хеширования

Номер	Имя переменной	Тип
1	*ASC*	…
2		
3	*CAR*	…
4	*DOG*	…
5	*CAB*	…
6	*EGG*	…
7		

. . . . . .

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.

Обычно выход из конфликтов осуществляется циклической проверкой одного 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, вы-

полняемое 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

$$n, rehash(n), rehash^2(n), \ldots, rehash^{p-1}(n)$$

все являются адресами и

$$rehash^p(n) = n.$$

Описанная выше функция перехеширования определяется следующей 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 мы 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, если h – 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.

7.2.1 Сцепление элементов

В этом случае переполнения таблиц можно избежать путем использования указателей (рис. 7.1).

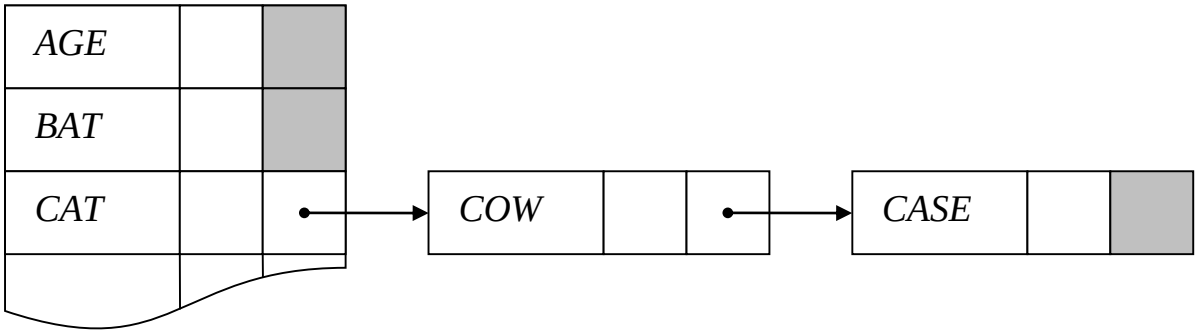


Рисунок 7.1 – Сцепление элементов

Для этого в таблице необходимо предусмотреть место для указателей, что ведет к увеличению объема последней.

7.2.2 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.

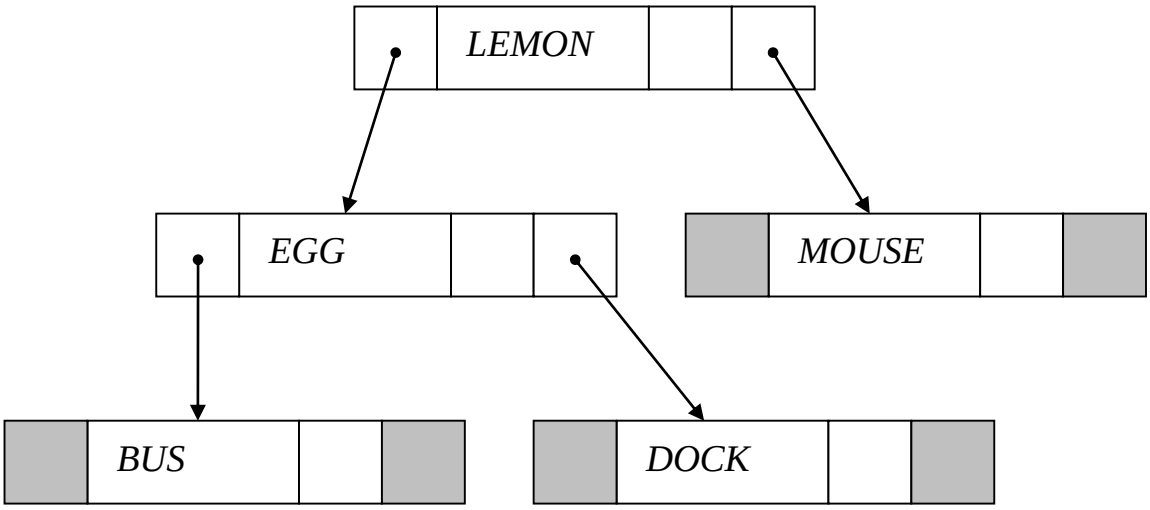


Рисунок 7.2 – Бинарное дерево

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.

7.3 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 [2]:

1) *основные виды*, или *атомарные*, например **int**, **real**, **char**, **bool** и др.;
2) *длинные и короткие виды*, или *модификаторы размера*, которые содержат символы **long** или **short**, появляющиеся перед основными видами (существуют и другие модификаторы типа);
3) *указатели* на адрес ячейки памяти, выделенной для данного вида;
4) *структурные виды*, типа **struct** с 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., используемыми 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 представлен структурой с отдельными указателями на список параметров и результат (рис. 7.3).

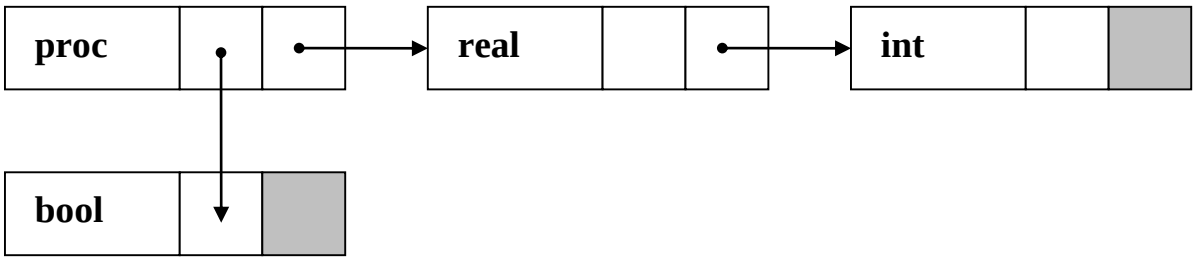
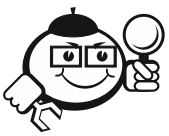


Рисунок 7.3 – Структура процедуры



Аналогичным образом, описатель

struct(**int** i, **struct**(**int j**, **bool** y), **real** r)

может быть представлен так, как показано на рис. 7.4.

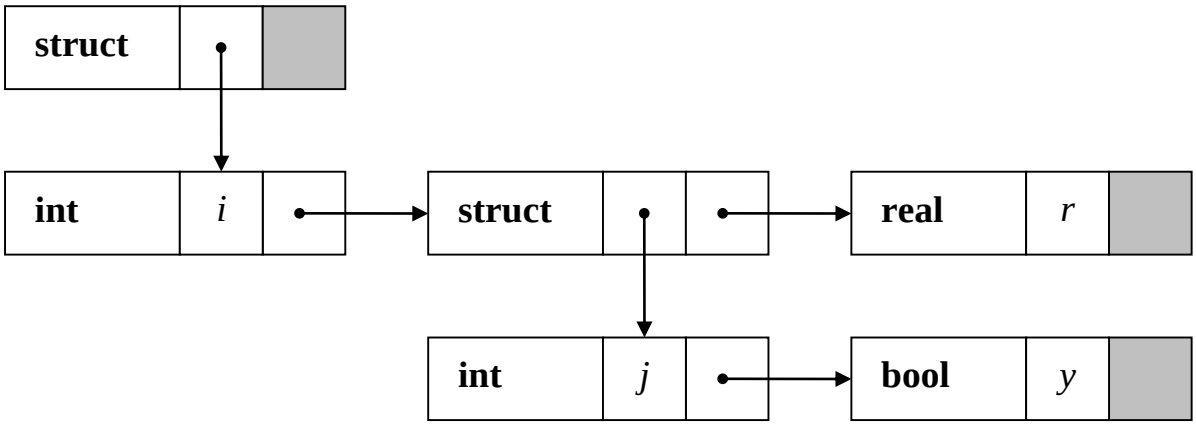


Рисунок 7.4 – Структура типа **struct**

. . . . . .

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.

7.4 Распределение памяти

7.4.1 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 заключается в отображении зна-

чений, 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 [2].

Каждой программе для хранения значений переменных и промежуточных значений выражений необходим определенный объем памяти.

Например, если идентификатор описывается как

int *x*,

т.е. *x* может принимать значение целого типа, 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 как

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 на размер типа 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 одномерный массив. Если считать, что массив имеет левую и правую стороны, память может выделяться слева направо. При этом применяется указатель, показывающий первый свободный элемент массива.

Пример

Например, в результате описания

INTEGER *A*, *B*, *X*, *Y*

память выделяется так, как это показано на рис. 7.5.

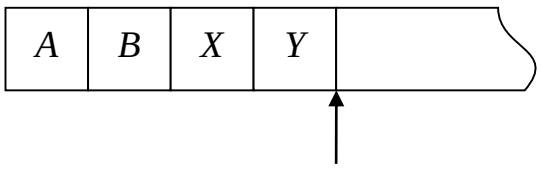


Рисунок 7.5 – Распределение памяти для Фортрана

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.

Пример

Пусть имеется программа вида:

```
begin
    real x, y
    …
    begin
        int c, d
        …
    end
    …
    begin
        char p, q
        …
    end
    …
end
```

На рис. 7.6 показаны «моментальные снимки» стека времени прогона на различных этапах ее выполнения. На данном рисунке изображено место, занимаемое значениями идентификаторов во время прогона.

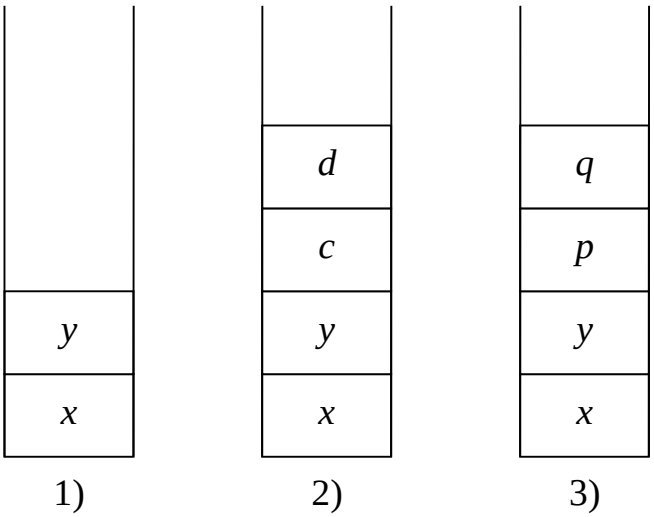


Рисунок 7.6 – Стек времени прогона

. . . . . .

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. 7.7). Это упрощает настройку указателя при выходе из блока.

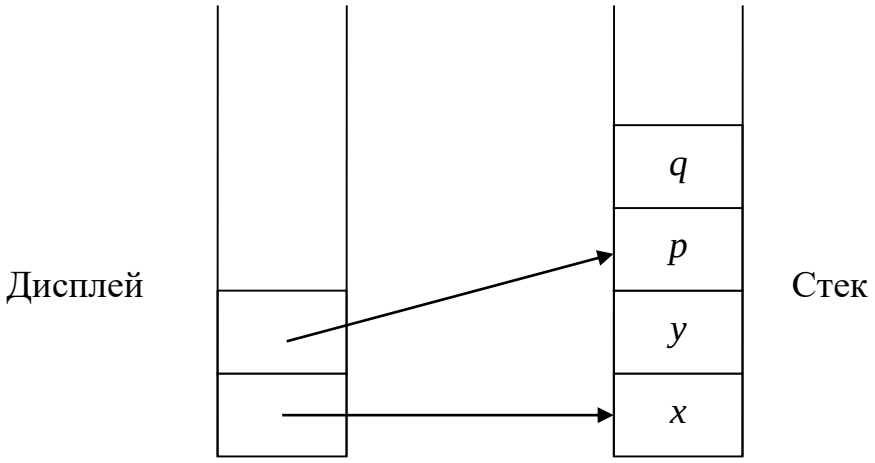


Рисунок 7.7 – Система «дисплей-стек»

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.


Пример

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 статической 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. 7.8).

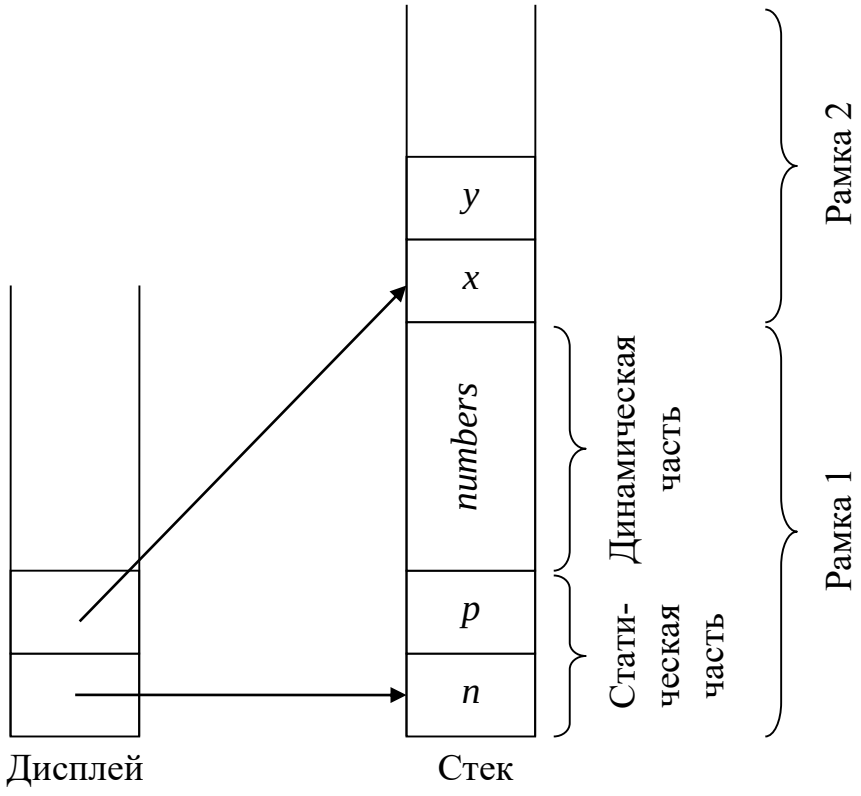


Рисунок 7.8 – Стек прогона для массива

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 по-

требоваться. 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 части. Кроме информации о границах, в статической части может храниться указатель на сами элементы массива (рис. 7.9).

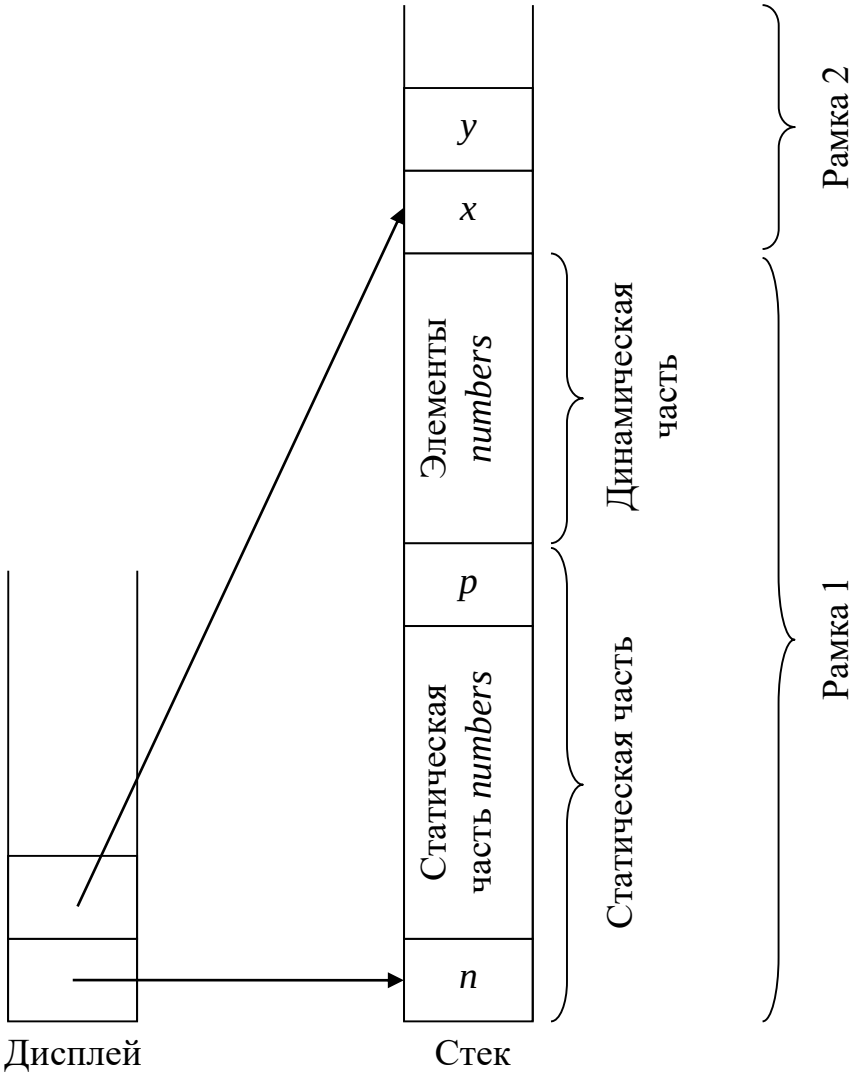


Рисунок 7.9 – Модифицированный стек прогона для массива

Когда в программе выбирается конкретный элемент массива, его адрес внутри динамической памяти должен вычисляться в процессе прогона.

Пример

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:

$$\text{ADDR}(table[I, J]) = \text{ADDR}(table[l_1, l_2]) + (u_2 - l_2 + 1)*(I - l_1) + (J - l_2).$$

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 $(u_2 - l_2 + 1)$ задает число различных значений, которые может принимать i-й индекс. Расстояние между элементами, отличающимися на единицу в i-м индексе, называется i-м *шагом* и обозначается s_i.

Пример шагов массива приведен на рис. 7.10.

int *N*[1:5, 1:5, 1:5]

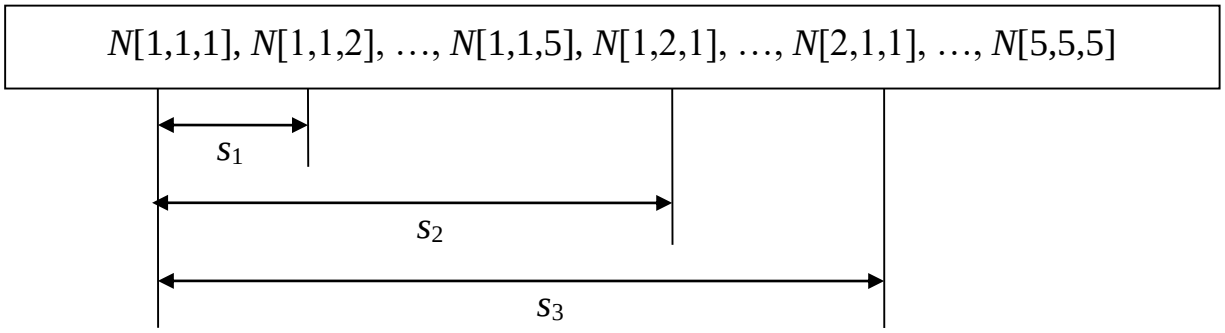


Рисунок 7.10 – Схема смещений

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 . . . . . .

Рассмотрим выражение

$$x := a+b\times c.$$

При вычислении выражения 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.

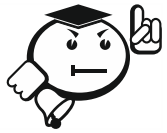
. . . . . .

Динамическая память должна распределяться во время прогона, статическая же – распределяется во время компиляции. Объем статической рабо-

чей памяти, который должен выделяться каждой рамке, определяется не рабочей памятью, требуемой в конце блока, а *максимальной* рабочей памятью, требуемой в любой точке внутри блока. Для статической рабочей памяти эту величину можно установить в процессе компиляции.

7.4.2 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 появ-

ляется 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.

7.4.3 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. 7.11).

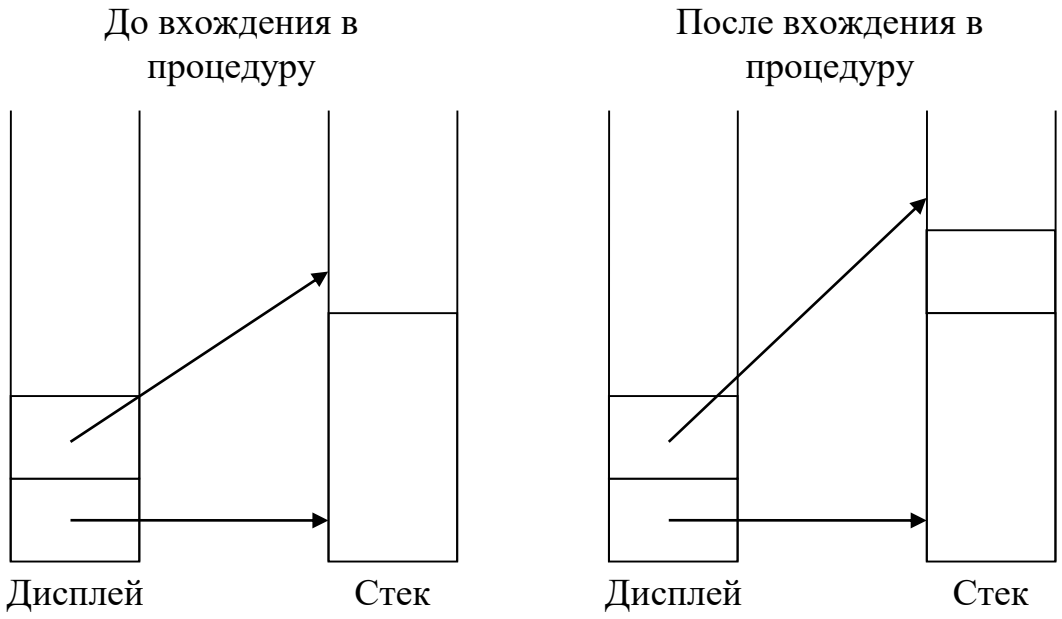


Рисунок 7.11 – Изменение дисплея при входе в процедуру

Конечно, после выхода из процедуры дисплей должен быть восстановлен.

Один из способов связи между фактическими и формальными параметрами заключается во 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 смо-

гут 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.

7.4.4 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 так, как это сделано на рис. 7.12.

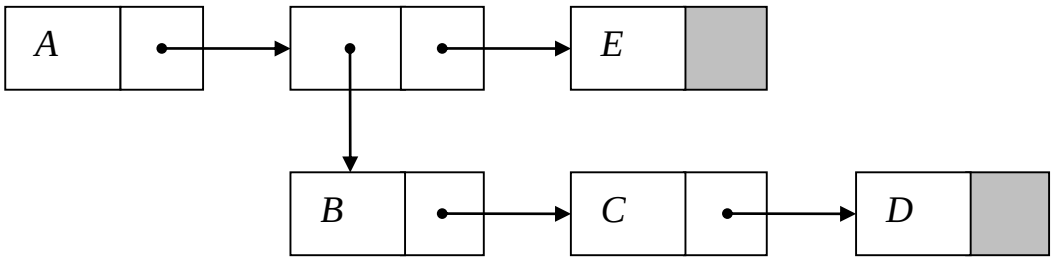


Рисунок 7.12 – Список

Каждый список состоит из головной части – литеры или указателя на другой список и хвостовой – указателя на другой список или нулевого списка. Список, изображенный на рис. 7.12, можно записать в следующем 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. 7.13).

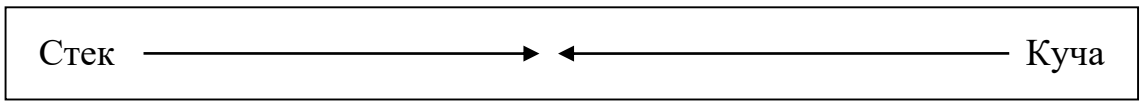
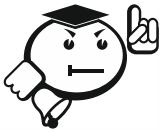


Рисунок 7.13 – Структура распределения памяти

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*.

Счетчик ссылок. При использовании счетчика ссылок память восстанавливается сразу после того, как она оказывается недоступной для про-

граммы. 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, указывающих на эту ячейку. Счетчики ссылок обновляются во время выполнения программы, и когда значение конкретного счетчика становится нулем, соответствующий объем памяти можно восстанавливать.

Пример

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 рис. 7.14.

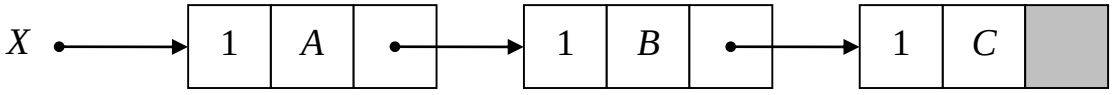


Рисунок 7.14 – Счетчики ссылок

Результат присвоения переменной *Y* указателя на второй элемент списка изображен на рис. 7.15.

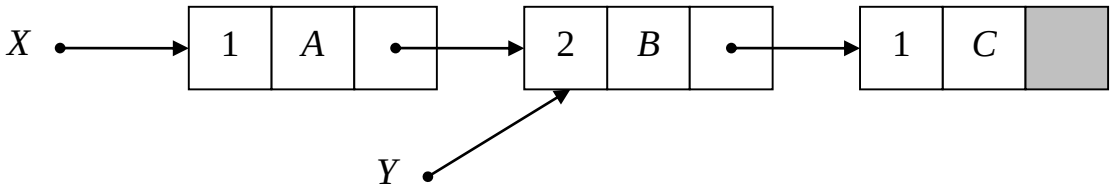


Рисунок 7.15 – Счетчики ссылок после первого присвоения

Результат последующего присвоения *X* = NULL приведен на рис. 7.16.

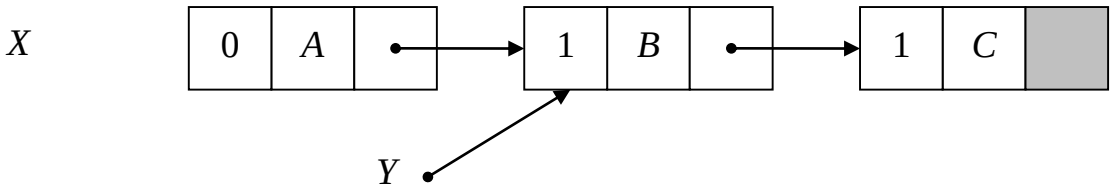


Рисунок 7.16 – Счетчики ссылок после второго присвоения

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:

1. Уменьшить на единицу счетчик ссылок ячейки, на которую указывал идентификатор правой части присвоения.
2. 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 будет доступа ни к одному из объемов памяти.

Пример

Это четко видно на примере циклического списка (рис. 7.17). Ни один из его счетчиков не является нулем, хотя никакие указатели на него извне не указывают. Этот объем памяти не восстановится никогда.

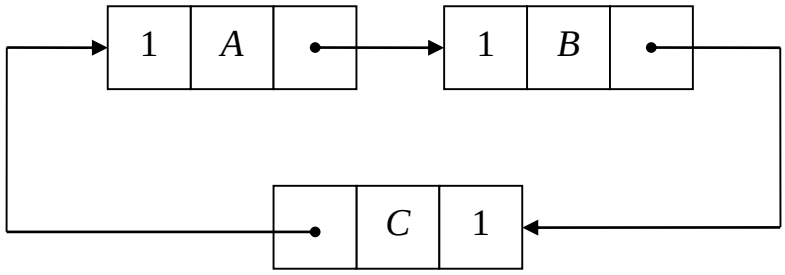


Рисунок 7.17 – Циклический список

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*».

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:

1. *Фаза маркировки*. Все адреса (или ячейки), к которым могут обращаться идентификаторы, имеющиеся в программе, маркируются путем изменения бита либо в самой ячейке, либо в отображении памяти в другом месте.

2. *Фаза уплотнения*. Все маркированные ячейки передвигаются в один конец кучи (дальний от стека).

Фаза уплотнения не тривиальна, так как она может повлечь за собой изменение указателей. Однако в общем случае она алгоритмически достаточно проста.

Самым критическим фактором является объем рабочей памяти, имеющийся у сборщика мусора. Будет нелогично, если самому сборщику мусора потребуется большой объем памяти. Кроме того, желательно, 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

Пример

До вызова этой процедуры куча могла иметь вид табл. 7.2, 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 7.2 – Состояние кучи

A	*left*	*right*	*mark*
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)
        fi
      fi
    od
  od
end
```

Пример

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. 7.18).

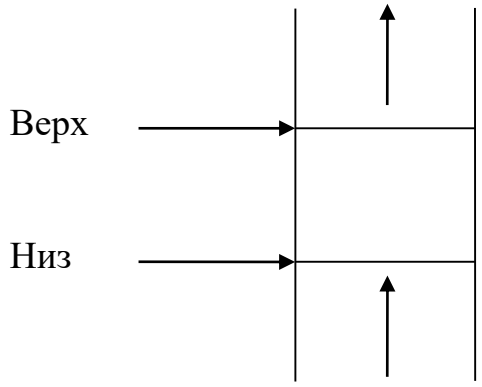


Рисунок 7.18 – Стек с двумя указателями

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 алгоритм 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.

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. 7.19.

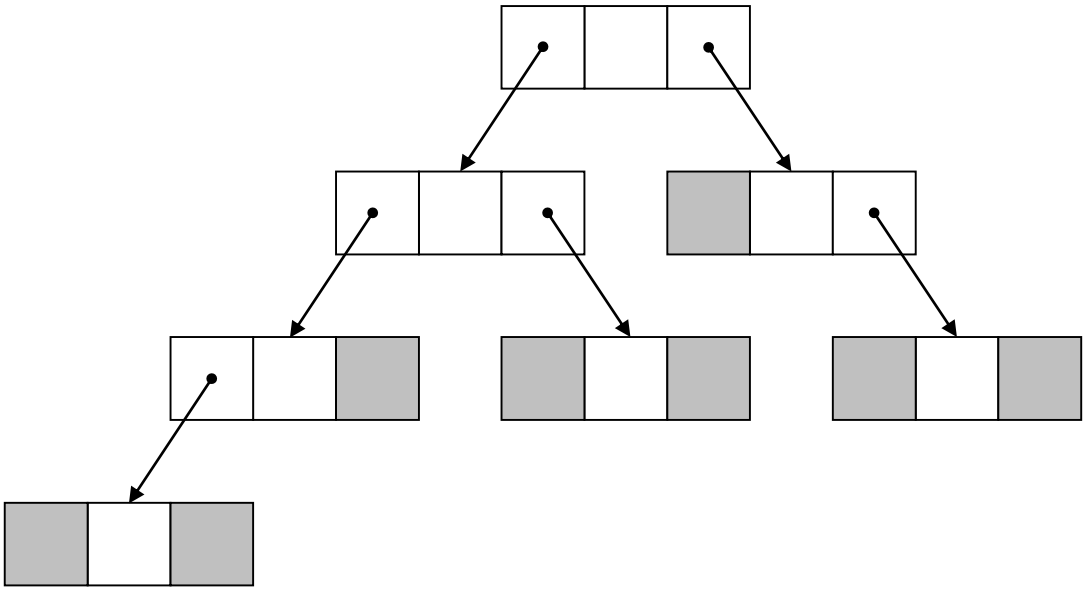


Рисунок 7.19 – Бинарное дерево

К тому моменту времени, когда фаза маркировки достигнет нижней левой ячейки, это дерево будет выглядеть так, как изображено на рис. 7.20.

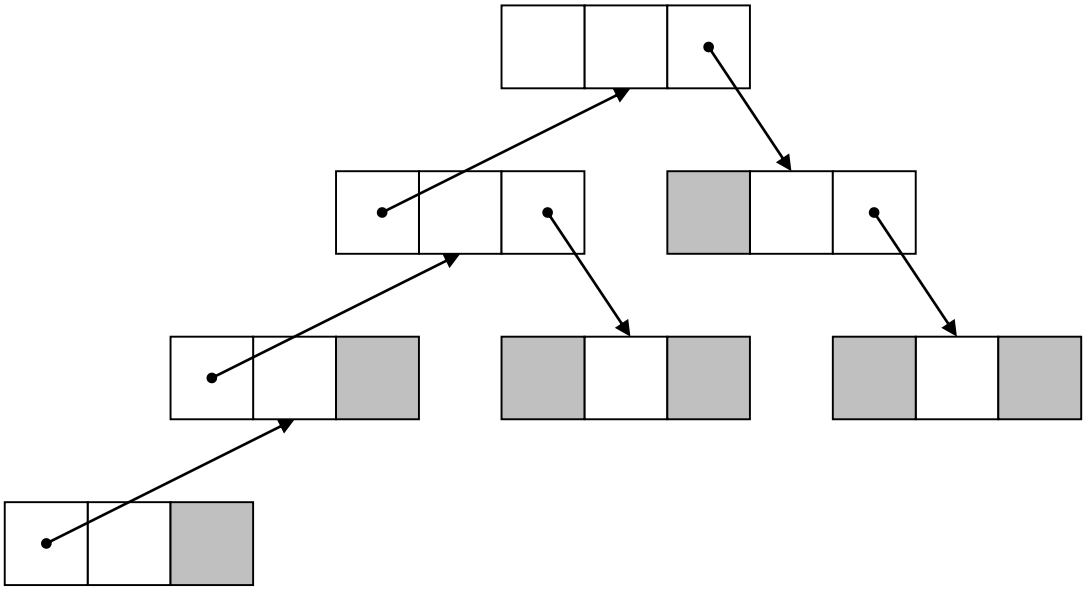


Рисунок 7.20 – Бинарное дерево в конце обхода

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 по главе 7

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.
8. Стек времени прогона, структура и программная реализация.
9. Технология распределения статической и динамической части стека.

10. Методы вызова параметров в процедурах, их достоинства и недостатки.
11. Организация распределения памяти при выполнении процедур.
12. Организация памяти для структур типа список (куча).
13. Освобождение памяти в куче. Счетчик ссылок.
14. Сборка мусора. Алгоритмы *MARK*1 и *MARK*2. Их преимущества и недостатки.
15. Сборка мусора. Алгоритмы Кнута, Шорра и Уэйта.

8 Генерация кода

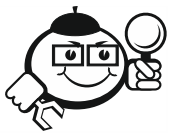
8.1 Генерация промежуточного кода

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 [2]:

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 являются *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 высокого 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 [5].

Пример

Выражение

$$a + b + c \times d$$

можно представить в виде четверок:

$$a + b = 1$$

$$c \times d = 2$$

$$1 + 2 = 3$$

и в 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.

Пример

Например, присвоение

$$a := b$$

в виде четверки представляется как

$$a := b = 1,$$

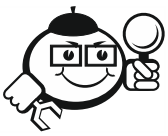
а в виде тройки – как

$$a := b.$$

Не менее популярны в качестве промежуточного кода *префиксные* и *постфиксные* нотации [4]. Например, инфиксное выражение $a + b$ в префиксной нотации имеет вид $+ a\ b$, а в постфиксной – $a\ b\ +$.

Таким образом, в ***префиксной нотации*** *каждый знак операции появляется перед своими операндами, а в* ***постфиксной нотации*** *– после. Префиксная нотация известна также как* ***польская запись****, а постфиксная – как* ***обратная польская запись*** **(*запись Лукашевича*)**.

В префиксной и постфиксной нотациях 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.



Пример

Например, выражение

$$(a + b)\times(c + d)$$

в префиксной форме записывается следующим образом:

$$\times + a\ b + c\ d,$$

а в постфиксной так:

$$a\ b + c\ d + \times.$$

Преобразование привычной инфиксной записи выражений в постфиксную запись 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.

Пример

Рассмотрим бинарное дерево выражения $(a + b)\times c + d$ (рис. 8.1).

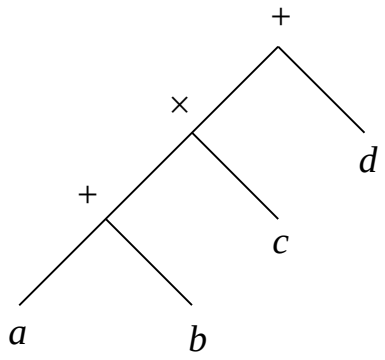


Рисунок 8.1 – Бинарное дерево

Чтобы получить представление префиксного выражения дерево обходят сверху в порядке:

– посещение корня;
– обход левого поддерева сверху;
– обход правого поддерева сверху,

что дает + × + *a b c d*.

Для получения постфиксного представления дерево обходят снизу:

– обход левого поддерева снизу;
– обход правого поддерева снизу;
– посещение корня,

в результате чего имеем $a\ b + c \times d +$.

. . . . . .

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

ASSIGN *type*, *addr*1, *addr*2

для присваивания. Тип необходим как параметр, чтобы определить размер значения, передаваемого из *addr*1 в *addr*2.

8.2 Структура данных для генерации кода

Для хранения адресов операндов на время, пока их нельзя выдать как параметры промежуточного кода, необходим стек значений [2]. 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:

$$EXP \to TERM \mid EXP + TERM \mid EXP - TERM$$

$$TERM \to FACT \mid TERM \times FACT \mid TERM / FACT$$

$$FACT \to constant \mid identifier \mid (EXP)$$

Для данных правил после чтения идентификатора или константы, знака операции и второго операнда необходимо выполнить следующие действия:

A_1. После чтения идентификатора или константы (т.е. листа синтаксического дерева) поместить в нижний стек соответствующие характеристики.

A_2. После чтения оператора поместить символ операции в стек знаков операций.

A_3. 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:

$$EXP \to TERM \mid EXP + \langle A_2 \rangle\ TERM\ \langle A_3 \rangle \mid EXP - \langle A_2 \rangle\ TERM\ \langle A_3 \rangle$$

$$TERM \to FACT \mid TERM \times \langle A_2 \rangle\ FACT\ \langle A_3 \rangle \mid TERM / \langle A_2 \rangle\ FACT\ \langle A_3 \rangle$$

$$FACT \to constant\ \langle A_1 \rangle \mid identifier\ \langle A_1 \rangle \mid (EXP)$$

Нижний стек частично используется для передачи информации о типе по синтаксическому дереву.

Пример

Рассмотрим дерево для $a \times b + x \times y$ (рис. 8.2).

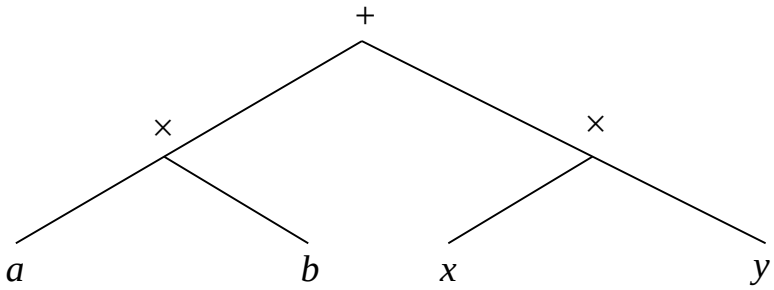


Рисунок 8.2 – Синтаксическое дерево для арифметического выражения

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 A_1, A_2 и A_3 в том порядке, в каком они будут возникать при трансляции выражения:

$$a\ \langle A_1\rangle \times \langle A_2\rangle\ b\ \langle A_1\rangle\ \langle A_3\rangle + \langle A_2\rangle\ x\ \langle A_1\rangle \times \langle A_2\rangle\ y\ \langle A_1\rangle\ \langle A_3\rangle\ \langle A_3\rangle$$

Каждый вызов A_3 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:

A_1. Поместить статические характеристики a в нижний стек.

A_2. Поместить знак «×» в стек знаков.

A_1. Поместить статические характеристики b в нижний стек.

A_3. 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.

A_2. Поместить знак «+» в стек знаков.

A_1. Поместить статические характеристики x в нижний стек.

A_2. Поместить знак «×» в стек знаков операций.

A_1. Поместить статическую характеристику *y* в нижний стек.

A_3. 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.

A_3. 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 A_1, A_2, A_3 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. 8.1.)

Таблица 8.1 – 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.

8.3 Генерация кода для типичных конструкций

8.3.1 Присвоение

. . . . . .

В процессе ***присвоения***

destination := *source*

значение, соответствующее ***источнику*** (*source*)*, присваивается значению* ***получателя*** (*destination*)*, которое является адресом*:

p := *x* + *y* (значение *x* + *y* присвоить *p*)

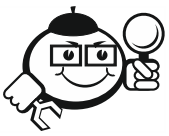
. . . . . .

Допустим, что статические характеристики источника и получателя находятся в вершине нижнего стека. Последовательность действий: из ниж-

него 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. Например, если источник – целое число, то его можно сначала преобразовать в вещественное число, 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).


Пример

Например, в программе

```
begin
    pointer real xx
    begin
        real x
        xx := x
    end
end
```

присвоение недопустимо, и это может быть обнаружено во время компиляции, если в таблице символов или в нижнем стеке имеется информация об области действия.

Для присвоения генерируется код, 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.

8.3.2 Условные зависимости

. . . . . .

Условная зависимость *имеет вид*

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:

$$\text{CONDITIONAL} \rightarrow \textbf{if}\ B\ \langle A_1\rangle\ \textbf{then}\ C\ \langle A_2\rangle\ \textbf{else}\ D\ \langle A_3\rangle$$

Действия A_1, A_2, A_3 означают (*next* – значение номера следующей метки, присваиваемое компилятором):

A_1. 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.

A_2. Генерировать код для перехода через ветвь **else** (то есть переход к концу условной зависимости):

GOTO *L<next>*

Удалить из стека номер метки (помещенный в стек действием A_1), назвать ее *i*, генерировать код для размещения метки:

SET LABEL *L<i>*

Поместить в стек значение *next*. Увеличить *next* на 1.

A_3. 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.

8.3.3 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.

8.3.4 Циклы

Определим простейший ***цикл*** *как*

for *i* := 1 **to** 10 **do** <*something*>

. . . . . .

Для генерации кода требуется четыре действия, которые размещаются следующим образом:

for *i* := 1 ⟨A_1⟩ **to** 10 ⟨A_2⟩ **do** ⟨A_3⟩ <something> ⟨A_4⟩

Эти действия таковы:

A_1. Выделить память для управляющей переменной *i*. Поместить сначала в эту память 1:

MOVE 1, address(<*controlled variable*>)

Здесь <*controlled variable*> – управляющая переменная цикла.

A_2. Генерировать код для записи в память значения верхнего предела рабочего стека:

MOVE address (<*ulimit*>) <*wostack*>,
<*current block number*>, <*wostack pointer*>,

где <*wostack pointer*> – указатель рабочего стека. Увеличить указатель рабочего стека и уменьшить указатель нижнего стека, где хранились статические характеристики верхнего предела.

A_3. 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.

A_4. Генерировать код для увеличения управляющей переменной

PLUS address(<*controlled variable*>), 1

Удалить из стека номер (*i*). Генерировать код для перехода к *L*<*i*>:

GOTO *L*<*i*>

Удалить из стека номер метки (*j*). Поместить метку в конец цикла:

SET LABEL *L*<*j*>

Таким образом, цикл

for *i* := 1 **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*>

GOTO *L*1

SET LABEL *L*2

Действия A_4 можно видоизменять, если приращение управляющей переменной будет не стандартным (1), а иным:

for *i* := 0 **by** 5 **to** 10 **do** <*something*>

Для этого придется вычислять приращение и хранить его значение в рабочем стеке, чтобы использовать как приращение.

Если цикл содержит часть **while**, например

for *i* := 1 **to** 10 **while** *a* < *b* **do**,

то действие A_3 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.

8.3.5 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:

1. 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.
2. Поместить в стек <*idstack pointer*>. Поместить в стек <*wostack pointer*>. Поместить в стек <*block number*>. Все эти значения ссылаются на включающий блок и могут потребоваться вновь после того, как будет покинут блок, в который только что осуществлен вход:

   <*idstack pointer*> := 0

   <*wostack pointer*>:= 0
3. Генерировать код для направления DISPLAY

   BLOCK ENTRY <*block number*>
4. Увеличить номер уровня блока на 1. Увеличить *gbn* (наибольший использованный до сих пор номер блока) на 1 и присвоить это значение номеру блока.
5. Прочитать информацию о видах и добавить ее в таблицу видов (если язык использует сложные виды (структуры)).

При выходе из блока необходимо выполнить следующие действия:

1. Обновить таблицу блоков, задав размер стека идентификаторов и т.п. для покинутого блока.
2. Исключить информацию в таблице символов для покинутого блока.
3. Генерировать код для изменения дисплея:

   BLOCK EXIT <*block number*>
4. Удалить из стека <*block number*>. Удалить из стека <*wostack pointer*>. Удалить из стека <*idstack pointer*>. Уменьшить уровень блока на 1.
5. Поместить результат (при необходимости) в рамку стека вызывающего блока.

8.3.6 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.

8.4 Проблемы, связанные с типами

Основной проблемой для трансляторов с языков высокого уровня является *приведение* (автоматическое изменение) *типов*. Здесь можно выделить, как минимум, шесть задач [2]:

1) *Распроцедуривание*, например переход от **procedure real** к **real**.
2) *Разыменование*, например переход от **pointer real** к **real**.
3) *Объединение*, например переход от **real** к **struct**(**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:

$$\begin{aligned} SOFT \rightarrow\ & deprocedure \mid \\ & deprocedure\ SOFT \end{aligned}$$

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:

$$\begin{aligned} MEEK \rightarrow\ & deprocedure \mid \\ & deprocedure\ MEEK \mid \\ & dereference \mid \\ & dereference\ MEEK \end{aligned}$$

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:

$$\begin{aligned} STRONG \rightarrow & dereference\ STRONG \mid \\ & deprocedure\ STRONG \mid \end{aligned}$$

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 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 находится в крепкой синтаксической позиции, которая не допускает расширения. Однако внутри выбирающего предложения один вариант допускает сильное приведение, что может повлечь за собой расширение. В этом случае объектным видом окажется **real**, и первый вариант следует расширить, а второй 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.

8.5 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 [2].

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. То есть в некоторых случаях требуется выполнить действие (переслать значение по новому адресу).

Для повышения эффективности выдаваемого кода при компиляции можно проделать дополнительную работу, которую называют ***оптимизацией****.*

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 по главе 8

1. Технология создания промежуточного кода. Виды промежуточного кода.
2. Алгоритм преобразования арифметического выражения в префиксную и постфиксную формы.
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.

9 Исправление и диагностика ошибок

9.1 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 будут удовлетворены таким подходом, поскольку они ожидают от компилятора [2]:

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 конструкций языка и забывает описать идентификатор или согласовать открывающую скобку с закрывающей и т.д.
– Программист неправильно пишет слово языка или какой-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 др.), 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.

9.2 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 [2]:

- Одна из литер оказывается недействительной, т.е. она не может быть включена ни в одну из лексем. 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 ни 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, например,

string *food* := ”BREED

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 не обязательно действительную последовательность символов). Для правильных в лексическом смысле программ это не представляет трудностей. При лексически неправильных программах приходится или игнорировать последова-

тельность символов, или включать дополнительные. Может потребоваться изменение написания символов, разбиение 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.

9.3 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 [2]:

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 скобок 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 хватает 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

Рассмотрим участок кода

if *b* **then** *x* **else** (*p*+*q*×*r*^2 **fi**.

Здесь пропущена закрывающая скобка. Это не обнаружится до тех пор, пока не встретится **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.

9.4 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 [2].

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.

9.4.1 Методы исправления синтаксических ошибок

Режим переполоха. Один из 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. Метод *исключения символов* 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.

 Пример

Например, в записи

c := *d*+3; **end**,

где символ «;» является недопустимым, 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 *включения символов* – подстановки одного из таких символов перед недопустимым символом, который вызвал ошибку.

 Пример

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. Один 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.

 Пример

Так, можно обращаться, например, с ошибками типа «;» перед **end** или пропуск «;».

9.4.2 Предупреждения

Наряду с сообщениями о синтаксических ошибках, анализатор может выдавать ***предупреждения****, когда ему встретилась допустимая, но маловероятная последовательность символов.*

 Пример

Пример маловероятной последовательности символов:

; **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.

9.4.3 Сообщения о синтаксических ошибках

Всякий раз при обнаружении анализатором синтаксической ошибки должно печататься соответствующее сообщение. Например,

SYNTAX ERROR IN LINE 22.

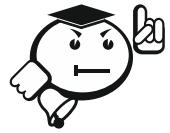
Или местоположение ошибки может описываться полнее:

SYNTAX ERROR IN 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 берутся 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.

9.5 Контекстно-зависимые ошибки

Некоторые конструкции типичных языков программирования нельзя описать с помощью контекстно-свободной грамматики [2]. Например, с точки зрения таблицы разбора, программы с неописанными идентификаторами синтаксически правильны.

Контекстно-зависимые ошибки *– это ошибки, которые 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 приводить 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* ALREADY 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.

9.6 Ошибки, связанные с употреблением типов

Правила, определяющие, где в программе возможно появление различных значений, не являются контекстно-независимыми [2]. 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.

Пусть в программе целому идентификатору присваивается литеральное значение:

```
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.

9.7 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 [2]:

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 которого возникла аварийная ситуация.

9.8 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 [2]:

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 по главе 9

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.

Заключение

В данном пособии были рассмотрены проблемы теоретического описания языков программирования и методов трансляции. Можно сделать вывод, что все изученные методы синтаксического анализа (такие, как конечные автоматы, регулярные выражения, распознаватели и грамматики) позволяют, в принципе, решать одинаковый круг задач. Во-первых, они определяют формальный язык, т.е. показывают, как, используя символы алфавита языка, построить правильные предложения на данном языке. Во-вторых, с их помощью для любого формального языка можно построить множество допустимых программ (в более общем случае – предложений или цепочек). Если это множество бесконечно, т.е. в языке присутствуют рекурсивные конструкции, делающие возможными предложения неограниченной длины, то можно построить лишь подмножество правильных цепочек, например ограниченной длины. В-третьих, для любых предложений или цепочек можно сделать вывод о том, являются ли они верными с точки зрения языка.

Если программа прошла синтаксическую проверку и разбита на лексемы, можно приступать к генерации объектного кода. Это уже задачи компилятора – сгенерировать и оптимизировать код, организовать распределение памяти и т.д. Далее из промежуточного объектного кода при помощи программ-сборщиков (от англ. linker) получается исполняемый код, но данный вопрос уже выходит за рамки изложенного материала.

Список литературы

1. Ахо А., Ульяман Дж. Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1978. – 612 с.
2. Хантер 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. – 128 с.
4. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. – М.: Мир, 1979. – 656 с.
5. Вайнгартен Ф. Трансляция языков программирования. – М.: Мир, 1977. – 192 с.
6. Гросс М., Лантен А. Теория формальных грамматик. – М.: Мир, 1971. – 294 с.

Глоссарий

Автомат с магазинной памятью – конечный автомат, в котором запоминающее устройство организовано по магазинному принципу (стек).

Алгоритм – конечное число команд, каждая из которых может выполняться механически за фиксированное время и с фиксированными затратами.

Алфавит – любое множество символов, не обязательно конечное и даже счетное.

Альтернативные порождающие правила – правила с одинаковой левой частью.

Вывод – процесс замены в цепочке подцепочки, являющейся левой частью порождающего правила, его правой частью.

Грамматика – порождающая система для описания языка.

Грамматика общего вида – грамматика без ограничений на правила.

Грамматика LL(k) – грамматика, в которой входная цепочка разбирается слева направо, используются выводы из левых частей правил к правым, а вариант порождающего правила выбирается при помощи предварительного просмотра k символов входной цепочки.

Грамматика LR(k) – грамматика, в которой входная цепочка разбирается слева направо, свертка правил идет справа налево, а вариант порождающего правила выбирается при помощи предварительного просмотра k символов входной цепочки.

Дерево вывода – дерево цепочек, принадлежащих языку, выводимых из стартового символа КС-грамматики.

Диаграмма переходов конечного автомата – неупорядоченный помеченный граф переходов между состояниями автомата.

Драйвер – модуль компилятора, занимающийся семантическим разбором.

Защита от дурака – принцип, согласно которому транслятор должен быть построен так, что никакая цепочка не может нарушить его работоспособности, т.е. он должен реагировать на любые из них

Иерархия Хомского – классификация грамматик.

Конечные автоматы – один из методов задания языка программирования.

Контекстно-зависимая грамматика – грамматика, цепочка левой части каждого правила которой не длиннее цепочки правой части данного правила.

Контекстно-свободная грамматика (*КС-грамматика*) – грамматика, в левой части каждого правила которой имеется лишь один нетерминал.

Конфигурация автомата – композиция текущего состояния конечного автомата и цепочки непрочитанных символов во входной ленте.

Лексема – цепочка терминальных символов, с которой мы связываем некоторую лексическую структуру.

Лексическая ошибка – ситуация, когда лексический анализатор не в состоянии сгруппировать последовательность символов в лексему.

Лексический анализатор – транслятор, входом которого служит цепочка символов, представляющая программу, а выходом – последовательность лексем.

Направляющий символ – символ, определяющий выбор одной из альтернатив порождающего правила при выводе.

Нетерминальный символ – служебный символ для порождения слов языка.

Обратная польская запись – постфиксная нотация, в которой знак операции ставится после операндов.

Оптимизация кода – попытка сделать объектные программы более эффективными, т.е. быстрее работающими или более компактными.

Перевод – процесс преобразования цепочек одного алфавита в цепочки другого алфавита.

Польская запись – префиксная нотация, в которой знак операции ставится перед операндами.

Последующий символ – символ, появляющийся при выводе после данной подцепочки.

Правило – описание процесса порождения цепочек языка.

Праволинейная грамматика – КС-грамматика, правая часть каждого правила которой начинается с цепочки терминальных символов.

Предикат – утверждение, состоящее из нескольких переменных и принимающее значение 0 или 1 («ложь» или «истина»).

Предупреждение – ситуация, когда синтаксическому анализатору встретилась допустимая, но маловероятная последовательность символов.

Предшествующий символ (*символ-предшественник*) – символ, появляющийся в начале данной цепочки в процессе вывода.

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

Проход – фаза процесса компиляции, требующая полного прочтения исходного текста программы.

Распознаватель – частичный алгоритм, определяющий язык.

Регулярное множество – множество цепочек, порождаемых регулярным выражением.

Регулярное выражение – один из методов задания языка программирования.

Семантическое отображение – отображение структурированного входа в выход, который обычно является программой на машинном языке.

Символ – элемент алфавита.

Синтаксическая ошибка – ошибка, обнаруживаемая контекстно-свободным синтаксическим анализатором.

Синтаксический анализ – разбор, в котором исследуется цепочка лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным в синтаксисе языка.

Синтаксический управляемый перевод – метод построения промежуточного кода по синтаксическому дереву.

Синтаксическое отображение – связывание с каждым входом (программой на исходном языке) некоторой структуры, которая служит аргументом семантического отображения.

Таблица разбора – составная часть модуля семантического разбора, получаемая из грамматики.

Терминальный символ – символ, входящий в слова (цепочки) определяемого языка.

Тройка – двухадресный промежуточный код.

Уравнения с регулярными коэффициентами – уравнения, коэффициентами которых являются регулярные выражения.

Цепочка – последовательность символов алфавита.

Четверка – трехадресный промежуточный код.

Язык – множество цепочек в алфавите.