Министерство образования и науки Российской Федерации
ГОУ ВПО "Алтайский государственный технический
университет им. И.И.Ползунова"

Е.Н. КРЮЧКОВА

МЕТОДЫ АНАЛИЗА В ТЕОРИИ ФОРМАЛЬНЫХ ЯЗЫКОВ

Учебное пособие

Барнаул 2013

УДК 681.142.2:517

Крючкова Е.Н. Методы анализа в теории формальных языков. Анализ: Учебное пособие / Алт. госуд. технич. ун-т им. И.И.Ползунова. Барнаул, 2013. — 276с.

Учебное пособие предназначено для студентов вузов, специализирующихся в области программирования, в частности, для бакалавров по направлению 231000 — "Программная инженерия направлению 010300 Математика – Компьютерные науки.

Рекомендовано на заседании кафедры прикладной математики
протокол N 5 от 14 января 2013 г.

Рецензенты: Зав.кафедрой информатики С.И. Жилин (АлтГУ)
Профессор кафедры прикладной математики Л.И. Сучкова (АлтГТУ),

ВВЕДЕНИЕ

Данная книга представляет собой учебное пособие по курсу анализа формальных языков. Цель этого учебника – выяснить и проанализировать некоторые фундаментальные понятия, алгоритмы и методы, лежащие в основе теории и практики построения интерпретаторов, компиляторов и других видов процессоров, предназначенных для обработки формальных языков. В этом курсе можно выделить следующие основные разделы: структура языков программирования, лексический, синтаксический и семантический анализ.

Материал данного учебного пособия содержит достаточно полную теорию, необходимую для построения лексического и синтаксического блоков компилятора, а также блока семантического контроля. Теоретический материал сопровождается практическими методами программирования соответствующих блоков.

С момента своего появления языки программирования, являясь средством общения человека с компьютером, представляют обширную и важную область прикладной математики. Имеются две категории программистов, непосредственно связанных с языками программирования. К первой из них относятся все программисты без исключения — лица, использующие в профессиональной деятельности языки программирования для описания алгоритмов решения задач на ЭВМ. В первую очередь такие специалисты заинтересованы в том, чтобы языки программирования были легкими в изучении и в понимании, содержали необходимые средства описания алгоритмов решения прикладных задач. Ко второй категории относятся специалисты по системному программному обеспечению ЭВМ, которых обычно называют системными программистами. Одной из часто решаемых такими специалистами задач является разработка и описание удобных языков общения человека с компьютером, а также реализация компиляторов или интерпретаторов для этих языков. Естественно, что такая деятельность требует профессионального владения способами описания и реализации языков программирования. Однако следует отметить, что и прикладной программист, хорошо представляющий процессы, происходящие на уровне компиляции его программы, имеет более высокий уровень профессионализма и обладает знаниями, необходимыми как для более быстрого и качественного написания программ, так и для их отладки.

Развитие теории и практики построения компиляторов шло параллельно с развитием языков программирования. Первым компилятором, который давал эффективный объектный код, был компилятор с Фортрана (Бэкус и др., 1957 год). С тех пор были написаны многочисленные компиляторы, интерпретаторы, генераторы и другие программы, обрабатывающие текстовую информацию определенной синтаксической структуры. Задачей компилятора является перевод программы с языка программирования в последовательность машинных команд, выполняющую те действия, которые предполагал программист. Структура языка программирования и способ его описания оказывают основополагающее влияние на способ проектирования компилятора. Процесс компиляции можно рассматривать как взаимодействие нескольких процессов, которые определяются структурой исходного языка. Современные методы проектирования языковых процессоров различных типов базируются на методах теории формальных грамматик, языков и автоматов. Фактически все методы разработки компиляторов основаны на использовании автоматных и контекстно-свободных грамматик, а синтаксический анализатор, выполняющий грамматический разбор, является ядром любого компилятора.

В данном учебнике можно выделить пять основных разделов, не вполне соответствующих номерам глав:

1) теория языков программирования (главы 1-3);
2) лексический анализ (глава 4);
3) методы синтаксического анализа языков (главы 6 — 9);
4) методы семантического контроля языков программирования (глава 5);
5) методы нейтрализации ошибок (глава 10).

Первые разделы посвящены теории формальных языков, их описанию на основе лексики, синтаксиса и семантики. Знание структуры языка программирования позволяет перейти к проектированию структуры транслятора с этого языка. В последующих главах рассматриваются особенности блоков, из которых построен транслятор, а также алгоритмы и методы их реализации.

Язык программирования определяется, с одной стороны, лексикой и синтаксисом языка, а, с другой стороны, семантическими аспектами каждой отдельно взятой программы. Проблема лексического и синтаксического анализа является теоретически решенной. Имеется общее понимание того, как можно синтаксически определить язык, а программу на этом языке синтаксически обработать. Что же касается семантических аспектов языка, то здесь еще многое нужно выяснить как на теоретическом, так и на практическом уровне.

Алгоритмы и методы трансляции мы будем строить на основе формальных моделей. Если на пути от формальной модели к ее реализации на реальной машине выполнены все предписанные алгоритмом действия со всей требуемой аккуратностью, то полученная Вами реализация соответствующей программы будет свободна от ошибок (или по крайней мере от большого их количества). Для профессиональных программистов это особенно важно.

В связи с ограниченностью объема пособия, примеры, иллюстрирующие основные понятия предмета, вынесены большей частью в упражнения и задачи. Поэтому для более полного усвоения материала необходимо уделить время реализации помещенных в конце каждой главы заданий.

Каждая глава заканчивается тестами для самостоятельной оценки знаний студентами. Среди перечисленных вариантов возможных ответов или утверждений необходимо выбрать один правильный ответ на вопрос или верное утверждение.

Студентам, желающим более подробно познакомиться с теорией алгоритмических языков и методов компиляции, можно порекомендовать книги, список которых приведен в конце пособия.

Введем обозначения, которые будем использовать в книге. Одно из основных математических понятий, которое используется для описания языков, — это контекстно–свободные грамматики (КС–грамматики).

Порождающей грамматикой называется упорядоченная четверка

$$G = (V_T, V_N, P, S), \text{ где}$$

V_T — конечный алфавит, определяющий множество терминальных символов;

V_N — конечный алфавит, определяющий множество нетерминальных символов;

P — конечное множество правил вывода — множество пар вида $u \to v$, где $u, v \in (V_T \cup V_N)^*$;

S — начальный нетерминальный символ — аксиома грамматики, $S \in V_N$.

Бесконтекстные или контекстно–свободные грамматики (КС–грамматики) — это грамматики, правила вывода которых имеют вид $A \to \phi$, где $\phi \in (V_T \cup V_N)^*$, $A \in V_N$.

Грамматика, не содержащая правил с пустой правой частью, называется неукорачивающей грамматикой.

Как правило, если не оговорено особо, прописные буквы латинского алфавита обозначают нетерминальные символы грамматики, а строчные буквы латинского алфавита — терминальные символы. Стандартный символ S обычно обозначает аксиому КС–грамматики.

Алфавит — непустое конечное множество. Элементы алфавита называются символами. Цепочка над алфавитом $\Sigma = \{a_1, a_2, ..., a_n\}$ есть конечная последовательность элементов a_i. Длина цепочки x — число ее элементов, обозначается $|x|$. Цепочка нулевой длины называется пустой цепочкой и обычно обозначается ε. Непустой называется цепочка ненулевой длины.

Пусть L и M — языки над некоторым алфавитом. Произведение языков есть множество $LM = \{xy | x \in L, y \in M\}$. В частности, $\{\varepsilon\}L = L\{\varepsilon\} = L$. Используя понятие произведения, определим итерацию L^* и усеченную итерацию L^+ множества L:

$$L^+ = \bigcup_{i=1}^{\infty} L^i,$$

$$L^* = \bigcup_{i=0}^{\infty} L^i,$$

где степени языка L можно рекурсивно определить следующим образом:

$$L^0 = \{\varepsilon\},\ L^1 = L,\ L^{n+1} = L^n L.$$

Например, пусть $L = \{a\}$, тогда

$$L^* = \{\varepsilon, a, aa, aaa, \dots\},$$

$$L^+ = \{a, aa, aaa, ...\} .$$

В частном случае в качестве языка может выступать сам алфавит. Множество всех цепочек (включая пустую цепочку ε) над алфавитом Σ обозначается через Σ^* . Множество всех цепочек кроме пустой цепочки ε над алфавитом Σ обозначается через Σ^+ . Например, $\Sigma = \{a, b\}$, тогда $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$, $\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, ...\}$.

Конечным автоматом называется автомат без рабочей ленты, имеющий входную ленту, с которой за один такт может быть прочитан один входной символ. Конечный автомат — это шестерка вида

$$A = (K, \Sigma, \delta, p_0, F),\ \text{где}$$

K — конечное множество состояний,

Σ — алфавит,

δ — функция переходов, в общем случае — недетерминированное отображение $\delta : K \times \Sigma \to 2^K$,

p_0 — начальное состояние, $p_0 \in K$,

F — множество заключительных состояний, $F \subseteq K$.

Частным случаем конечных автоматов являются детерминированные конечные автоматы с функцией переходов $\delta : K \times \Sigma \to K$.

Известно, что для произвольного недетерминированного конечного автомата можно построить эквивалентный детерминированный.

В отличие от конечного автомата автомат с магазинной памятью (МП–автомат) имеет рабочую ленту — магазин. МП–автомат — это семерка вида

$$M = (K, \Sigma, \Gamma, \delta, p_0, F, B_0), \text{ где}$$

K — конечное множество состояний,

Σ — алфавит,

Γ — алфавит магазина,

δ — функция переходов, $\delta : K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \to 2^{K \times \Gamma^*}$,

p_0 — начальное состояние,

F — множество заключительных состояний,

B_0 — символ из Γ для обозначения маркера дна магазина.

В общем случае это определение соответствует недетерминированному автомату. В отличие от конечного автомата для произвольного недетерминированного МП–автомата нельзя построить эквивалентный детерминированный МП–автомат.

Будем использовать следующие теоретико-множественные обозначения: если A — некоторое множество объектов, то $\overline{A}$ обозначает дополнение A до N, т.е. $\overline{A} = N \setminus A$. $A = B$ означает, что A и B одинаковы как множества, т.е. A и B состоят из одних и тех же элементов; $x \in A$ означает, что x — элемент множества A. Обозначение $\{|\}$ указывает на образование множества: $\{x|...x...\}$ — это множество всех таких x, что выражение "$...x...$" верно для всех элементов этого множества.

Для заданных элементов x и y будем рассматривать упорядоченные пары $\langle x, y\rangle$, состоящие из элементов x и y, взятых именно в таком порядке. Аналогично будем использовать обозначение $\langle x_1, x_2, ..., x_n\rangle$ — для упорядоченной n–ки или кортежа длины n, состоящего из элементов x_1, x_2,..., x_n и именно в этом порядке. Через $A \times B$ обозначим декартово произведение множеств A и B, т.е. $A \times B = \{\langle x, y\rangle | x \in A \& y \in B\}$. Аналогично $A_1 \times A_2 \times ... \times A_n = \{\langle x_1, x_2, ..., x_n\rangle | x_1 \in A_1 \& x_2 \in A_2 \& ... \& x_n \in A_n\}$. Декартово произведение множества A на себя n раз обозначается A^n.

Другие общие и специальные обозначения будут вводиться по мере необходимости.

Глава 1

ФОРМАЛЬНЫЕ ГРАММАТИКИ И ЯЗЫКИ

1.1 Понятие порождающей грамматики и языка

Прежде, чем рассматривать формальное определение языка и грамматики, рассмотрим такое описание на интуитивном уровне. Язык можно определить как некоторое множество предложений заданной структуры, имеющих, как правило, некоторое значение или смысл. Правила, определяющие допустимые конструкции языка, составляют *синтаксис языка*. Значение (или смысл) фразы определяется *семантикой языка*. Например, по правилам синтаксиса языка Си фраза $(X * 2)$ является правильным выражением, в отличие от фразы $(2X*)$. Семантика языка Си определяет, что фраза

$$for(inti = 0; i < 10; i++)S[i]+ = A[i];$$

является оператором цикла, в котором переменная i последовательно принимает значения 0, 1,...,9.

Если бы все языки состояли из конечного числа предложений, то не ставилась бы и проблема описания синтаксиса, т.к. достаточно просто перечислить все допустимые предложения конечного языка — и язык задан. Но почти все языки содержат неограниченное (или очень большое) число правильно построенных фраз, поэтому возникает *проблема описания бесконечных языков с помощью конечных средств*. Различают порождающее и распознающее описание языка. *Порождающее описание языка* означает наличие алгоритма, который последовательно порождает все правильные предложения этого языка. Любая строка принадлежит языку тогда и только тогда, когда она появляется среди генерируемых строк. *Распознающее описание языка* означает наличие алгоритма, который для любой фразы может определить, принадлежит эта фраза языку или нет. Средством порождающего задания языка являются *грамматики*. Рассмотрим основные понятия, связанные с языками и порождающими грамматиками.

Алфавит — непустое конечное множество. Элементы алфавита называются символами. Цепочка над алфавитом $\Sigma = \{a_1, a_2, ..., a_n\}$ есть конечная последовательность элементов a_i. Длина цепочки x — число ее элементов, обозначается $|x|$. Цепочка нулевой длины называется пустой цепочкой и обычно обозначается ε. Непустой называется цепочка ненулевой длины.

Цепочки x и y равны, если они одинаковой длины и совпадают с точностью до порядка символов, из которых состоят, т.е. если $x = a_1a_2...a_n$, $y = b_1b_2...b_k$, то $n = k$ и для всех $1 \leq i \leq k$ справедливо равенство $a_i = b_i$.

Пусть $x = a_1a_2...a_n$ и $y = b_1b_2...b_k$ — некоторые цепочки. Конкатенацией (или сцеплением, или произведением) цепочек x и y называется цепочка $xy = a_1a_2...a_nb_1b_2...b_k$, полученная дописыванием символов цепочки y вслед за символами цепочки x. Например, если $x = cca, y = abba$, то $xy = ccaabba$. Поскольку ε — цепочка нулевой длины, то в соответствии с определением конкатенации можно написать $\varepsilon x = x\varepsilon = x$.

Множество всех цепочек (включая пустую цепочку ε) над алфавитом Σ обозначается через Σ^* . Например, $\Sigma = \{a, b\}$, тогда $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$. В дальнейшем мы увидим, почему принято обозначение Σ^*.

Язык L над алфавитом Σ есть некоторое множество цепочек над этим алфавитом, т.е. $L \subseteq \Sigma^*$. Необходимо различать пустой язык $L = \emptyset$ и язык, содержащий только пустую цепочку $L = \{\varepsilon\} \neq \emptyset$. Формальный язык L над алфавитом Σ — это язык, выделенный с помощью конечного множества некоторых формальных правил.

Пусть L и M — языки над алфавитом . Произведение языков есть множество $LM = \{xy|x \in L, y \in M\}$. В частности, $\{\varepsilon\}L = L\{\varepsilon\} = L$. Используя понятие произведения, определим итерацию L^* и усеченную итерацию L^+ множества L:

$$L^+ = \bigcup_{i=1}^{\infty} L^i,$$

$$L^* = \bigcup_{i=0}^{\infty} L^i,$$

где степени языка L можно рекурсивно определить следующим образом:

$$L^0 = \{\varepsilon\},\ L^1 = L,\ L^{n+1} = L^n L.$$

Например, пусть $L = \{a\}$, тогда

$$L^* = \{\varepsilon, a, aa, aaa, \dots\},$$

$$L^+ = \{a, aa, aaa, ...\}\ .$$

Определение 1.1. Порождающей грамматикой называется упорядоченная четверка

$$G = (V_T, V_N, P, S),\ \text{где}$$

V_T — конечный алфавит, определяющий множество терминальных символов;

V_N — конечный алфавит, опредедяющий множество нетерминальных символов;

P — конечное множество правил вывода — множество пар вида $u \to v$, где $u, v \in (V_T \cup V_N)^*$;

S — начальный нетерминальный символ — аксиома грамматики, $S \in V_N$.

Определение 1.2. В грамматике $G = (V_T, V_N, P, S)$ цепочка x непосредственно порождает цепочку y, если $x = \alpha u \beta$, $y = \alpha v \beta$ и $u \to v$ является правилом грамматики G, т.е. $u \to v \in P$. Говорят также, что y непосредственно выводится из x. Непосредственная выводимость y из x обозначается $x \Rightarrow y$.

Определение 1.3. В грамматике G цепочка y выводима из цепочки x, если существуют цепочки $x_0, x_1, .., x_k$ такие, что $x = x_0, y = x_k$ и для всех i ($1 \leq i \leq k$) $x_{i-1} \Rightarrow x_i$, т.е.

$$x = x_0 \Rightarrow x_1 \Rightarrow \ldots \Rightarrow x_k = y.$$

На каждом шаге вывода применяется одно правило грамматики. Число шагов в выводе цепочки y из цепочки x называется длиной вывода. Выводимость обозначается

$$x \overset{*}{\Rightarrow} y.$$

Определение 1.4. Языком, порождаемым грамматикой $G = (V_T, V_N, P, S)$, называется множество терминальных цепочек, выводимых в грамматике G из аксиомы:

$$L(G) = \{x | x \in V_T^*; S \overset{*}{\Rightarrow} x\}.$$

Пример 1.1. Пусть $G = (V_T, V_N, P, S)$, где $V_T = \{a, b\}, V_N = \{S\}, P = \{S \to aSb, S \to ab\}$. Тогда в грамматике существуют выводы

$$\begin{aligned}
&S \Rightarrow ab,\\
&S \Rightarrow aSb \Rightarrow aabb,\\
&S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb,\\
&....
\end{aligned}$$

Таким образом, $L(G) = \{a^n b^n | n > 0\}$.

Если множество правил приводится без специального указания множества нетерминалов и терминалов, то обычно предполагается, что грамматика содержит в точности те терминалы и нетерминалы, которые встречаются в правилах. Предполагается также, что правые части правил, левые части которых совпадают, можно записывать в одну строку с вертикальной чертой "| " в качестве разделителя. Тогда грамматику, рассмотренную в примере 1.1, можно задать следующим образом:

$$G : S \to aSb | ab.$$

Нетрудно видеть, что терминальные символы грамматики — это такие символы, из которых состоят цепочки языка, порождаемого грамматикой. В языках программирования терминалами являются фактически используемые в них слова и символы, такие, как for, $+$, $float$ и т.д. Нетерминальные символы являются вспомогательными символами, используемыми только в процессе вывода и не входящими в цепочки языка. Обычно нетерминалы предназначены для обозначения некоторых понятий, например, при определении языков программирования нетерминалами служат такие элементы, как ⟨программа⟩, ⟨оператор ⟩, ⟨выражение⟩ и т.д. В силу того, что все цепочки языка выводятся из аксиомы, аксиоме должно соответствовать основное определяемое понятие, в частности, для языков программирования таким нетерминалом может быть ⟨программа⟩.

Чтобы легче было различать нетерминальные и терминальные символы, примем соглашение обозначать терминалы маленькими буквами, а нетерминалы — большими буквами (или заключать в угловые скобки). Будем также считать, что аксиомой является символ, стоящий в левой части самого первого правила грамматики.

1.2 Классификация грамматик

Правила порождающих грамматик позволяют осуществлять самые разные преобразования строк. Определенные ограничения на вид правил позволяют выделить классы грамматик. Общепринятой является предложенная Н. Хомским следующая *система классификации грамматик.*

Определение 1.5. Грамматики типа 0 — это грамматики, на правила вывода которых не наложено никаких ограничений.

Например, правило грамматики типа 0 может иметь вид $aAbS \to SbaaS$.

Определение 1.6. Грамматики типа 1 — грамматики непосредственно составляющих или контекстно–зависимые грамматики — это грамматики, правила вывода которых имеют вид $xAy \to x\phi y$, где $A \in V_N; x, y, \phi \in (V_N \cup V_T)^*$.

Например, правило контекстно–зависимой грамматики может иметь вид $aAbc \to aaabc$ или $Aa \to Ba$.

Определение 1.7. Грамматики типа 2 — бесконтекстные или контекстно–свободные грамматики — это грамматики, правила вывода которых имеют вид $A \to \phi$, где $\phi \in (V_T \cup V_N)^*$, $A \in V_N$.

Например, следующая грамматика является контекстно–свободной:

$$G : A \to aAb|ab.$$

Иногда выделяют специальный подкласс класса контекстно–свободных грамматик (КС–грамматик) — металинейные грамматики, правила вывода которых имеют вид $A \to xBy$ или $A \to \phi$, $x, y, \phi \in V_T^*$, $A, B \in V_N$. Приведенная выше грамматика является металинейной.

Определение 1.8. Грамматики типа 3 — автоматные грамматики. Различают два типа автоматных грамматик: леволинейные и праволинейные. Леволинейные грамматики — грамматики, правила вывода которых имеют вид $A \to Ba$ или $A \to a$, где $a \in V_T, A, B \in V_N$. Праволинейные грамматики — это грамматики, правила вывода которых имеют вид $A \to aB$ или $A \to a$.

В дальнейшем мы в основном будем рассматривать контекстно–свободные грамматики (или КС–грамматики) и автоматные грамматики.

Пример 1.2. Язык $\{a^n b^n c^n, n \geq 0\}$ порождается следующей грамматикой:

$$\begin{array}{ll} G_0 : & S \to AB \\ & A \to aADb|\varepsilon \\ & Db \to bD \\ & DB \to Bc \\ & B \to \varepsilon. \end{array}$$

Это грамматика типа 0. Пример вывода цепочки $a^2b^2c^2$ имеет вид:

$$\begin{array}{ll} S & \Rightarrow AB \Rightarrow aADbB \Rightarrow aaADbDbB \Rightarrow aaDbDbB \Rightarrow aabDDbB \Rightarrow \\ & \Rightarrow aabDbDB \Rightarrow aabbDDB \Rightarrow aabbDBc \Rightarrow aabbBcc \Rightarrow aabbcc. \end{array}$$

Пример 1.3.
Язык $a^n, n > 0$ порождается леволинейной грамматикой

$$G_2 : S \to Sa|a.$$

Вывод цепочки a^3 имеет вид:

$$S \Rightarrow Sa \Rightarrow Saa \Rightarrow aaa.$$

Определение 1.9. Язык L называется языком типа i, если существует грамматика типа i, порождающая L.

В частности, язык является КС–языком, если существует КС–грамматика, его порождающая. Тогда, язык $a^n b^n$ (см. пример 1.1) является КС–языком, а язык a^n— автоматным языком (см. пример 1.3).

1.3 Основные свойства КС–языков и КС–грамматик

Поскольку любой язык — это некоторое множество цепочек, представляет интерес выполнение специальных языковых и обычных теоретико–множественных операций над языками. Рассмотрим операции пересечения, объединения, итерации, усеченной итерации и произведения, выполняемые над классами КС–языков.

Теорема 1.1. Семейство КС–языков замкнуто относительно операций объединения, произведения, итерации и усеченной итерации.

Доказательство. Язык является контекстно–свободным, если существует КС–грамматика, порождающая его. Пусть L_1 и L_2 — КС–языки, тогда существуют КС–грамматики

$$G_1 = (V_{T1}, V_{N1}, P_1, S_1) \text{ и } G_2 = (V_{T2}, V_{N2}, P_2, S_2)\ ,$$

порождающие соответственно L_1 и L_2. Всегда можно считать, что множества нетерминальных символов этих КС–грамматик не пересекаются и $V_{N1} \cap V_{N2} = \emptyset$. Рассмотрим КС–грамматику

$$G = (V_{T1} \cup V_{T2}, V_{N1} \cup V_{N2} \cup \{S | S \notin V_T \cup V_N\}, P, S),$$

где S — новый нетерминал, а P — множество правил, полученное объединением правил исходных грамматик и двух новых правил $S \to S_1 | S_2,$ т.е.

$$P = P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}.$$

Любой вывод в G имеет вид $S \Rightarrow S_1 \overset{*}{\Rightarrow} x$ или $S \Rightarrow S_2 \Rightarrow^* y$, причем $S_1 \overset{*}{\Rightarrow} x$ — это всегда вывод в G_1, а $S_2 \overset{*}{\Rightarrow} y$ — это всегда вывод в G_2, т.к. множества нетерминальных символов V_{N1} и V_{N2} не пересекаются. Тогда $L(G) = L(G_1) \cup L(G_2)$.

Аналогично можно показать, что язык $L_1 L_2$ порождается грамматикой G, множество правил которой содержит кроме P_1 и P_2 правило $S \to S_1 S_2$.

Можно также показать язык L_1^+ порождается грамматикой, множество правил которой, кроме правил P_1, содержит два дополнительных правила $S \to SS_1 | S_1$, а язык L_1^* — грамматикой с дополнительными правилами $S \to SS_1 | \varepsilon$. $\square$

Операции произведения, усеченной итерации, итерации и объединения позволяют легко строить КС–грамматики сложных языков через грамматики простых языков. Например, пусть требуется построить КС–грамматику, порождающую идентификаторы языка Си.

Идентификатор — это последовательность цифр и букв, начинающаяся с буквы, следовательно, идентификатор можно определить как произведение понятия <буква> и понятия <знаки>. Знаки — это произвольная последовательность букв и цифр, в том числе и пустая, следовательно <знаки> — итерация элемента <знак>, в качестве которого могут выступать буквы и цифры. Тогда получим грамматику

$$\begin{aligned} G:\quad & <\text{идентификатор}> \to <\text{буква}><\text{знаки}> \\ & <\text{знаки}> \to <\text{знак}><\text{знаки}> | \varepsilon \\ & <\text{знак}> \to <\text{буква}> | <\text{цифра}> \\ & <\text{буква}> \to A|B|...|Z|a|b|...|z| \\ & <\text{цифра}> \to 0|1|...|9 \end{aligned}$$

Рассмотрим операции дополнения и пересечения, выполняемые над КС–языками. Чтобы доказать незамкнутость семейства КС–языков относительно этих операций, сформулируем пока без доказательства следующую теорему (доказательство рассмотрим далее в параграфе 1.6).

Теорема 1.2. Язык $a^n b^n c^n, n > 0$ в алфавите {a,b,c} не является контекстно–свободным.

Теорема 1.3. Семейство КС–языков не замкнуто относительно операции пересечения.

Доказательство. Для доказательства достаточно привести хотя бы один пример таких КС–языков L_1 и L_2, пересечение которых не является КС–языком. Рассмотрим языки $L_1 = a^n b^m c^m, n, m > 0$ и $L_2 = a^n b^n c^m, n, m > 0$. Они являются контекстно–свободными, т.к. порождаются КС–грамматиками соответственно

$$\begin{aligned} G_1: \quad & S \to AB \\ & A \to aA|a \\ & B \to bBc|bc, \\ G_2: \quad & S \to AB \\ & A \to aAb|ab \\ & B \to cB|c. \end{aligned}$$

Пересечение L_1 и L_2 есть язык $a^n b^n c^n, n > 0$, по теореме 1.2. не являющийся контекстно–свободным.□

Теорема 1.4. Семейство КС–языков не замкнуто относительно операции дополнения.

Доказательство. Рассмотрим КС–языки L_1 и L_2. Пусть дополнение сохраняет свойства КС–языка оставаться контекстно–свободным. Тогда $\overline{L_1}$ и $\overline{L_2}$ — КС–языки. По теореме 1.1 объединение КС–языков $\overline{L_1}$ и $\overline{L_2}$ — КС–язык, т.е. $\overline{L_1} \cup \overline{L_2}$- КС–язык, но тогда и $\overline{\overline{L_1} \cup \overline{L_2}}$ — тоже КС–язык. Тогда $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$ — КС–язык для любых КС–языков L_1 и L_2, что противоречит теореме 1.3. □

Следствием доказанных теорем 1.3 и 1.4 является невозможность использования операций пересечения и дополнения при построении КС–грамматик сложных языков из КС–грамматик более простых языков в отличие от операций объединения, произведения, итерации и усеченной итерации.

Кроме правил использования операций объединения, произведения, итерации и усеченной итерации при синтезе КС–грамматик применяется еще одно правило, позволяющее сконструировать цепочки из симметрично стоящих элементов. Для того, чтобы грамматика порождала цепочки вида $x^n z y^n (n \geq 0)$, достаточно в ней иметь правила вида

$$A \to xAy|z.$$

Действительно, любой вывод из нетерминала A имеет вид $A \Rightarrow xAy \Rightarrow x^2 A y^2 \Rightarrow ... \Rightarrow x^n A y^n \Rightarrow x^x z y^n$.

Например, язык $a^n b^{3m} c^m a^{2n} = a^n (bbb)^m c^m (aa)^n$ ($n, m \geq 0$) порождается КС–грамматикой

$$\begin{aligned} G: \quad & S \to aSaa|B \\ & B \to bbbBc|\varepsilon. \end{aligned}$$

1.4 Грамматический разбор

В КС–грамматике может быть несколько выводов, эквивалентных в том смысле, что во всех них *применяются одни и те же правила к одним и тем же нетерминалам* в цепочках, полученных в процессе вывода, различие имеется только в порядке применения этих правил. Например, в грамматике

$$G: S \to ScS|b|a$$

возможны два эквивалентных вывода

$$\begin{aligned} S \Rightarrow ScS \Rightarrow Scb \Rightarrow acb \\ S \Rightarrow ScS \Rightarrow acS \Rightarrow acb. \end{aligned}$$

Можно ввести удобное графическое представление вывода, называемое деревом вывода, или деревом грамматического разбора, или синтаксическим деревом.

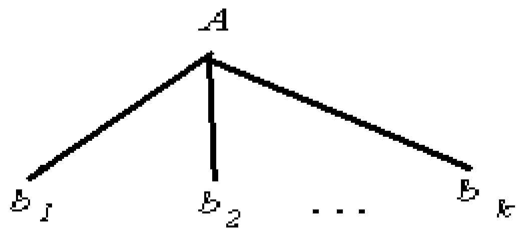
Определение 1.10. Деревом вывода цепочки x в КС–грамматике

$$G = (V_T, V_N, P, S)$$

называется упорядоченное дерево, каждая вершина которого помечена символом из множества $V_T \cup V_N \cup \{\varepsilon\}$ так, что каждому правилу

$$A \to b_1 b_2 \dots b_k \in P,$$

используемому при выводе цепочки x, в дереве вывода соответствует поддерево с корнем A и прямыми потомками b_1, b_2,..., b_k :

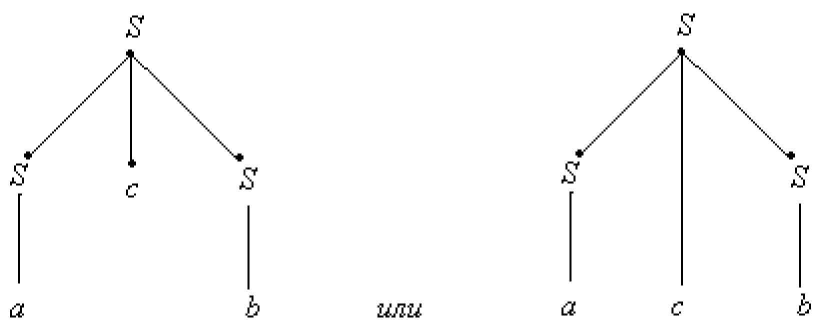


В силу того, что цепочка $x \in L(G)$ выводится из аксиомы S, корнем дерева вывода всегда является аксиома. Внутренние узлы дерева соответствуют нетерминальным символам грамматики. Концевые вершины дерева вывода (листья) — это вершины, не имеющие потомков. Такие вершины соответствуют либо терминалам, либо пустым символам ε при условии, что среди правил грамматики имеются правила с пустой правой частью. При чтении слева направо концевые вершины дерева вывода образуют цепочку, вывод которой представлен деревом. Именно по этой причине деревом вывода должно быть *упорядоченное* дерево.

Например, в рассмотренной выше грамматике с правилами

$$S \to ScS|b|a$$

цепочке acb соответствует дерево вывода

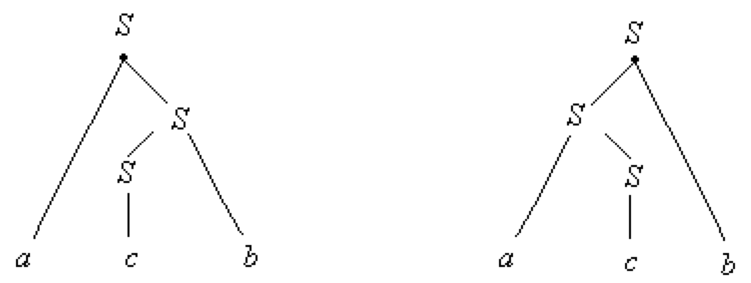


Дерево вывода часто называют также деревом грамматического разбора, деревом разбора, синтаксическим деревом. Процесс построения дерева разбора называется грамматическим разбором или синтаксическим анализом.

Одной цепочке языка может соответствовать более, чем одно дерево, т.к. она может иметь разные выводы, порождающие разные деревья. Например, в грамматике

$$G : S \to aS|Sb|c$$

цепочка $a$cb имеет два разных дерева вывода:



Определение 1.11. КС–грамматика G называется неоднозначной (или неопределенной), если существует цепочка $x \in L(G)$, имеющая два или более дерева вывода.

Если грамматика используется для определения языка программирования, желательно, чтобы она была однозначной. В противном случае программист и компилятор могут по–разному понять смысл некоторых программ. Рассмотрим, например, оператор if языка Паскаль. Пусть он порождается КС–грамматикой

$$G : S \to \text{ if } (V) \ S \text{ else } S| \text{ if } (V) \ S|O,$$

где V — соответствует понятию "выражение O — "оператор". Эта грамматика неоднозначна, т.к. существует оператор, имеющий два дерева вывода, представленных на рис. 1.1 и 1.2.

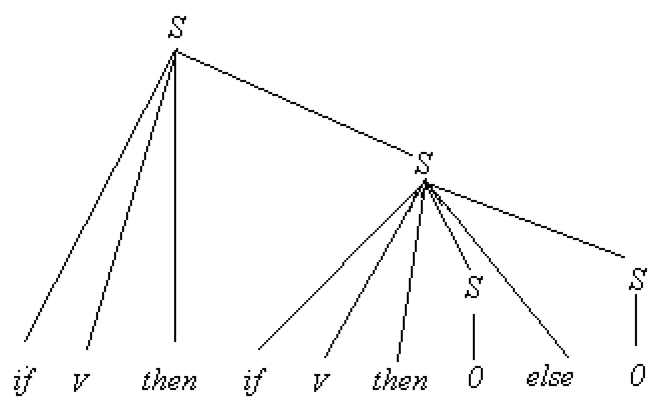


Рис. 1.1: Правильное дерево разбора оператора if.

Первое дерево предполагает интерпретацию if V then (if V then O else O), тогда как второе дерево дает if V then (if V then O) else O. Определенная нами неоднозначность — это свойство грамматики, а не языка. Для некоторых неоднозначных грамматик можно построить эквивалентные им однозначные грамматики. Например,

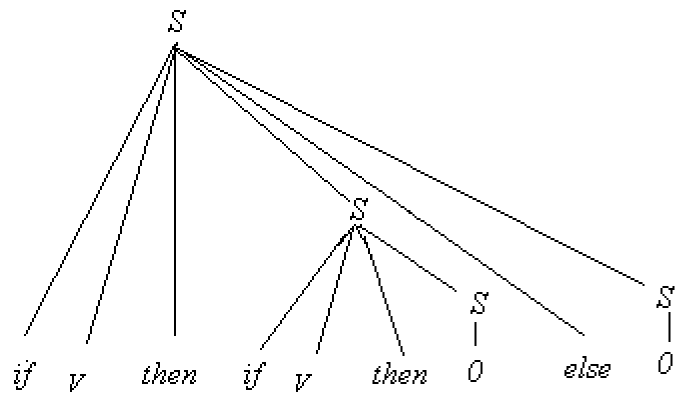


Рис. 1.2: Неправильное дерево разбора оператора if.

приведенная выше грамматика оператора IF неоднозначна потому, что ELSE можно ассоциировать с двумя разными THEN. Неоднозначность можно устранить, если договориться, что ELSE должно соответствовать последнему из предшествующих ему THEN:

$$\begin{aligned} G_1:\quad & S \rightarrow \text{ if } V \text{ then } S|\text{ if } V \text{ then } S1 \text{ else } S|O \\ & S_1 \rightarrow \text{ if } V \text{ then } S_1 \text{ else } S|O \end{aligned}$$

Рассмотренный выше оператор имеет в этой грамматике единственное дерево вывода, представленное на рис. 1.3 и являющееся аналогом дерева рис. 1.1. Аналог дерева 1.2 в данной грамматике построить невозможно.

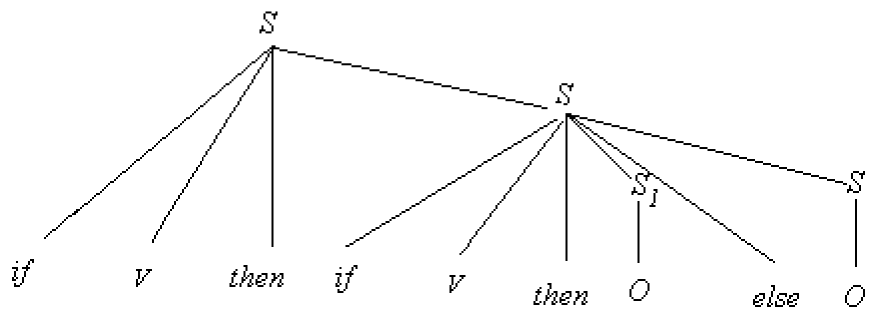


Рис. 1.3: Дерево разбора оператора if в однозначной КС–грамматике.

Это дерево предполагает интерпретацию if V then (if V then O else O). Попытка построить дерево вывода этой цепочки другим способом обречена на неудачу.

Определение 1.12. КС–язык называется существенно–неоднозначным (или существенно–неопределенным), если он не порождается никакой однозначной КС–грамматикой.

Процесс построения дерева вывода называется *грамматическим разбором.* Рассмотрим несколько определений, связанных с процессом построения дерева вывода.

Определение 1.13. Любая цепочка (не обязательно терминальная), выводимая из аксиомы, называется сентенциальной формой.

Например, в грамматике

$$G : S \to aSSb|abS|ab$$

существует вывод

$$S \Rightarrow aSSb \Rightarrow aabSSb \Rightarrow aababSb \Rightarrow aabababb,$$

тогда $S, aSSb, aabSSb, aababSb, aabababb$ — сентенциальные формы. Язык $L(G)$ составляют только терминальные сентенциальные формы.

Различают две стратегии разбора: восходящую ("снизу вверх") и нисходящую ("сверху вниз"). Эти термины соответствуют способу построения синтаксических деревьев. При нисходящей стратегии разбора дерево строится от корня (аксиомы) вниз к терминальным вершинам. Например, в грамматике

$$G : S \to aSSb|abS|ab$$

при нисходящем разборе цепочки $aabababb$ получаем последовательность частичных деревьев, представленную на рис. 1.4.

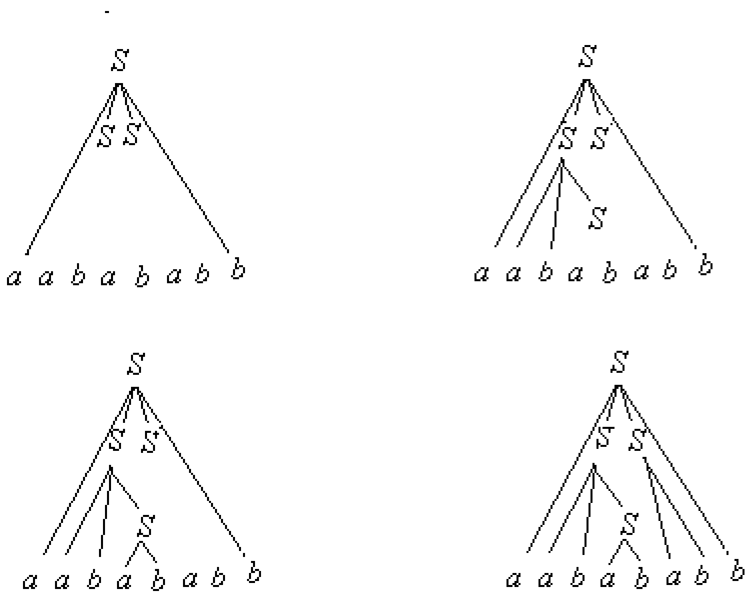


Рис. 1.4: Нисходящий разбор в КС-грамматике $G : S \to aSSb|abS|ab$

Главная задача при нисходящем разборе — выбор того правила $A \to \phi_i$ из совокупности правил $A \to \phi_1|\phi_2|\ldots|\phi_k$, которое следует применить на рассматриваемом шаге грамматического разбора.

При восходящем разборе дерево строится от терминальных вершин вверх к корню дерева — аксиоме. Например, дерево разбора рассмотренной цепочки $aabababb$ по восходящей стратегии представлено на рис. 1.5.

Определение 1.14. Преобразование цепочки, обратное порождению, называется приведением (или редукцией). Цепочка y прямо редуцируема к цепочке x в грамматике G, если x прямо порождает y.

Определение 1.15. Пусть $z = xty$ — сентенциальная форма, тогда t называется фразой сентенциальной формы z для нетерминального символа A, если $S \overset{*}{\Rightarrow} xAy$ и

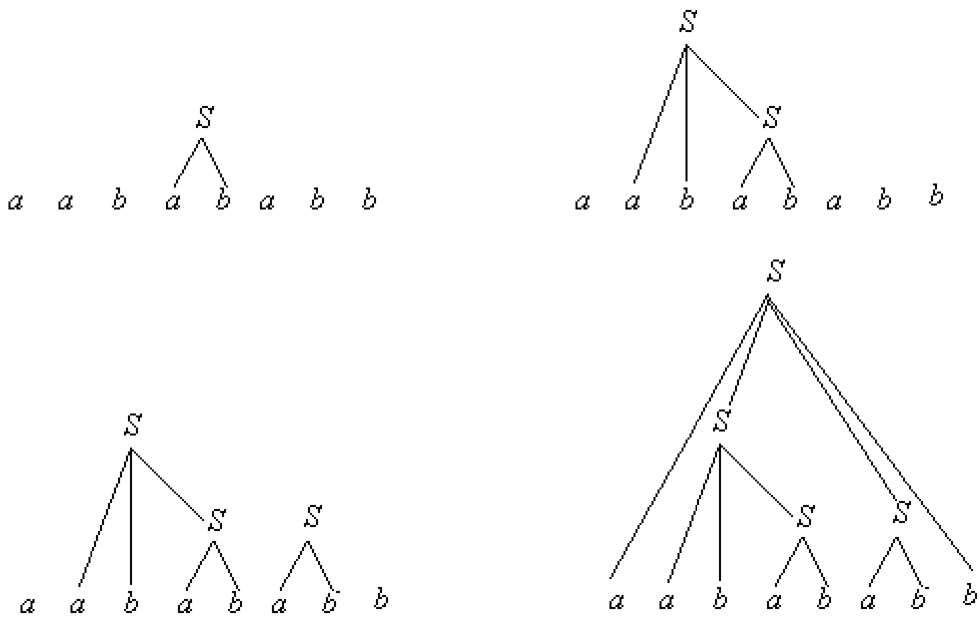


Рис. 1.5: Восходящий разбор в КС-грамматике $G : S \to aSSb|abS|ab$

$A \overset{*}{\Rightarrow} t$. Цепочка t называется простой фразой, если t является фразой и $A \to t$ — правило грамматики.

Обычно грамматический разбор выполняют слева направо, т.е. сначала обрабатывают самые левые символы рассматриваемой цепочки и продвигаются по цепочке вправо только тогда, когда это необходимо. При выполнении разбора по восходящей стратегии слева направо на каждом шаге редуцируется самая левая простая фраза. Самая левая простая фраза сентенциальной формы называется *основой.* Цепочка вправо от основы всегда содержит только терминальные символы.

Главная задача при восходящем разборе — поиск основы и, если грамматика содержит несколько правил с одинаковыми правыми частями, выбор нетерминального символа, к которому должна редуцироваться основа.

Если $x \notin L(G)$, то не существует вывода $S \overset{*}{\Rightarrow} x$ в грамматике G, а это значит, что для x нельзя построить дерево вывода. Любой компилятор выполняет синтаксический анализ исходного модуля и выдает сообщение об ошибке, если дерево разбора исходного модуля не существует.

Пример 1.4. Даны две КС–грамматики, порождающие выражение, которое может содержать знаки операций, круглые скобки и символы "a" в качестве операндов:

$$G_1 : S \to S + S|S - S|S * S|S/S|(S)|a,$$

$$\begin{array}{ll} G_2: & S \to S + A|S - A|A \\ & A \to A * B|A/B|B \\ & B \to a|(S). \end{array}$$

Требуется определить, являются ли эти грамматики однозначными. Если какая–либо из этих грамматик неоднозначна, привести пример цепочки, для которой существуют два различных дерева разбора.

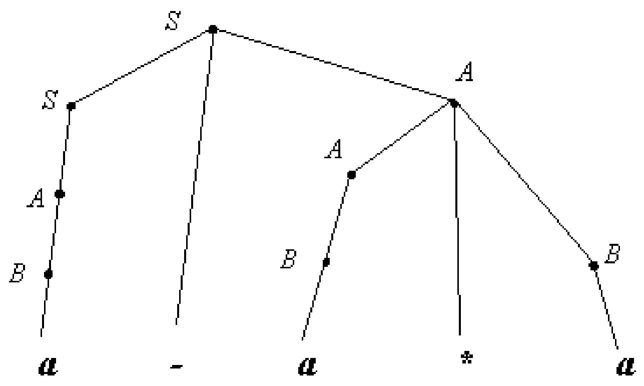
Очевидно, что в грамматике G_1 не учитываются приоритеты операций, поэтому любое выражение, содержащее знаки различных приоритетов, может быть выведено из аксиомы разными способами. Например, выражение $a - a * a$ можно вывести по–разному:

$$S \Rightarrow S - S \Rightarrow a - S \Rightarrow a - S * S \Rightarrow a - a * S \Rightarrow a - a * a,$$

$$S \Rightarrow S * S \Rightarrow S * a \Rightarrow S - S * a \Rightarrow a - S * a \Rightarrow a - a * a.$$

Этим выводам соответствуют разные деревья грамматического разбора.

Если первое из них отражает реальный порядок выполнения операций в языках программирования типа Си, то второе дерево противоречит таким правилам. Грамматика G_1 неоднозначна. В то же время в грамматике G_2 выражению $a - a * a$ соответствует единственное дерево разбора:



1.5 Преобразования КС–грамматик

Для любого КС–языка существует бесконечное число КС–грамматик, порождающих этот язык. Часто требуется изменить грамматику так, чтобы она удовлетворяла определеным требованиям, не изменяя при этом порождаемый грамматикой язык.

Определение 1.16. Грамматики G_1 и G_2 называются эквивалентными, если совпадают порождаемые ими языки.

1.5.1 Правила с одним нетерминалом

Рассмотрим первое эквивалентное преобразование — удаление правил вида "$<$ нетерминал $> \rightarrow <$ нетерминал $>$". Очевидно, что присутствие таких правил в грамматике существенно увеличивает высоту дерева разбора, а, следовательно, процесс разбора занимает больше времени.

Теорема 1.5. Для любой КС–грамматики можно построить эквивалентную грамматику, не содержащую правил вида $A \rightarrow B$, где A и B — нетерминальные символы.

Доказательство. Пусть имеется КС–грамматика

$$G = (V_T, V_N, P, S),$$

где $V_N = \{A_1, A_2, ..., A_n\}$, а в P входят правила указанного вида. Разобъем P на два непересекающихся подмножества:$P = P_1 \cup P_2$, $P_1 \cap P_2 = \emptyset$, где в P_1 включены все правила вида $A_i \rightarrow A_k$ $(A_i, A_k \in V_N)$, а в P_2 — все остальные правила, т.е. $P_2 = P \setminus P_1$. Определим для каждого $A_i \in V_N$ множество правил $P(A_i)$, включив в него все такие правила $A_i \to \phi$, что $A_i \stackrel{+}{\Rightarrow} A_j$ и $A_j \to \phi$ — правило из множества P_2. Построим новую КС–грамматику $G_1 = (V_T, V_N, P_1, S)$, в которой множество терминальных и нетерминальных символов и аксиома совпадают с соответствующими объектами грамматики G, а множество правил получено объединением правил множества P_2 и правил $P(A_i)$ для всех $1 \le i \le n$:

$$P = \bigcup_{i=1}^{n} P(A_i) \cup P_2.$$

Эта грамматика не содержит правил вида $A_i \to A_k$. Покажем, что она эквивалентна исходной, т.е. покажем справедливость $L(G) \subseteq L(G_1)$ и $L(G_1) \subseteq L(G)$. С этой целью рассмотрим вывод цепочки x в грамматике G. Рассмотрим самый левый шаг вывода, на котором применялось правило из P_1 (заметим, что если правила из P_1 не применялись, то вывод в G является одновременно и выводом в G_1). Цепочка x является терминальной в силу своей принадлежности языку $L(G)$, тогда существует шаг вывода, на котором применено правило из P_2. Но тогда в G_1 существует правило $A_i \to \phi \notin P_1$, т.е. часть вывода в G можно заменить на один шаг вывода в G_1. Продолжая эту процедуру через конечное число шагов (не более длины вывода) из вывода в G получим вывод x в G_1.Таким образом, показали $L(G) \subseteq L(G_1)$. Обратная процедура построения вывода в G по выводу в G_1 очевидна.$\square$

Доказательство теоремы 1.5. дает алгоритм эквивалентного преобразования грамматик с целью удаления правил вида $A \to B$.

Пример 1.5. Пусть задана КС–грамматика

$$\begin{aligned} G:\ \ & S \to aFb|A \\ & A \to aA|B \\ & B \to aSb|S \\ & F \to bc|bFc. \end{aligned}$$

Построим множества P_2, $P(S)$, $P(A)$, $P(B)$, $P(F)$:

1) $P_2 = \{S \to aFb, A \to aA, B \to aSb, F \to bc|bFc\}$;

2) $S \Rightarrow A \Rightarrow B$, т.е. $S \Rightarrow^* A, S \Rightarrow^* B$, тогда $P(A) = \{S \to aA, S \to aSb\}$;

3) $A \Rightarrow B \Rightarrow S$, т.е. $A \Rightarrow^* B, A \Rightarrow^* S$, тогда $P(A) = \{A \to aSb, A \to aFb\}$;

4) $B \Rightarrow S \Rightarrow A$, т.е. $B \Rightarrow^* S, B \Rightarrow^* A$, тогда $P(B) = \{B \to aFb, B \to aA\}$;

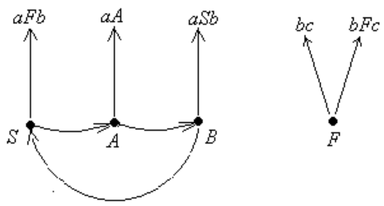
5) из F нельзя вывести нетерминальный символ, тогда $P(F) = \emptyset$.

Объединяя все эти правила, получаем грамматику, эквивалентную исходной:

$$\begin{aligned} G:\ \ & S \to aFb|aA|aSb \\ & A \to aA|aSb|aFb \\ & B \to aSb|aFb|aA \\ & F \to bs|bFc. \end{aligned}$$

Этот алгоритм удобно использовать при автоматизированном преобразовании грамматик с помощью ЭВМ. При неавтоматизированном преобразовании он оказывается довольно сложным. В таком случае проще применить графическую модификацию метода. С этой целью каждому нетерминальному символу и каждой правой части правил множества P_2 поставим в соответствие вершину графа. Из вершины с меткой U в вершину с меткой V направлено ребро, если существует в грамматике правило $U \to V$. Правило $A \to w$ принадлежит новой грамматике, если из вершины с меткой A существует путь в вершину с меткой w.

В нашем случае получим граф



1.5.2 Правила с одинаковыми правыми частями

При удалении правил вида $A \to B$ в множестве правил появляется много правил с одинаковой правой частью. Рассмотрим построение для произвольной КС–грамматики эквивалентной КС–грамматики, все правила которой имеют различные правые части.

Пусть $G_n = (V_T, V_N, P_n, S)$ — некоторая КС–грамматика, все нетерминальные символы которой имеют непустые несовпадающие множества нижних индексов:

$$V = \{A_{i_1 i_2 \ldots i_k}\}, i_n < i_m \text{ при } n < m.$$

Пусть в множестве P_n существуют правила с совпадающими правыми частями

$$\begin{array}{l} A_{i_1 i_2 \ldots i_k} \to \phi \ , \\ A_{j_1 j_2 \ldots, j_m} \to \phi \ , \\ A_{l_1 l_2 \ldots l_r} \to \phi \ . \end{array} \tag{1.1}$$

Построим грамматику G_{n+1} следующим образом:

1) заменим правила (1.1) на $A_{n_1 n_2 \ldots n_t} \to \phi$, где $n_i < n_j$ при $i < j$ и множество $\{n_1, n_2, \ldots n_t\}$ есть объединение индексов нетерминалов, стоящих в левых частях правил (1.1);

2) если некоторое A_i для $i \in \{n_1, n_2, \ldots, n_t\}$ есть аксиома S грамматики G_n, то новому множеству правил P_{n+1} принадлежит правило $S \to A_{n_1, n_2 \ldots n_t}$;

3) если множеству P_n принадлежат правила вида

$$B \to \xi A_{p_1 p_2 \ldots p_u} \eta, \tag{1.2}$$

такие, что $\{p_1, p_2, \ldots p_u\} \subset \{n_1, n_2, \ldots n_t\}$, то к новому множеству правил добавляются правила вида $B \to \xi A_{n_1 n_2 \ldots n_t} \eta$, полученные из (1.2) с помощью подстановки $A_{n_1 n_2 \ldots n_t}$ вместо некоторых (быть может, никаких) $A_{p_1 p_2 \ldots p_u}$ в правой части правила (1.2). Назовем такое построение G_{n+1} по G_n операцией частичного склеивания по индексам.

Пример 1.6. Пусть задана грамматика

$$\begin{array}{ll} G_1: & S \to bA_1 | baA_2 \\ & A_1 \to aaA_1 | ab | d \\ & A_2 \to aaA_2 | ab | c \end{array}$$

Выполним частичное склеивание, удалив правила $A_1 \to ab$ и $A_2 \to ab$:

$$\begin{array}{ll} G_2: & S \to bA_1 | baA_2 | bA_{12} | baA_{12} \\ & A_1 \to aaA_1 | d | aaA_{12} \\ & A_2 \to aaA_2 | c | aaA_{12} \\ & A_{12} \to ab \end{array}$$

Выполним опять частичное склеивание, удалив правила $A_1 \to aaA_{12}$ и $A_2 \to aaA_{12}$:

$$\begin{aligned} G_1: \quad & S \to bA_1|baA_2|bA_{12}|baA_{12} \\ & A_{12} \to ab|aaA_{12} \\ & A_1 \to aaA_1|d|aaA_{12} \\ & A_2 \to aaA_2|c|aaA_{12}. \end{aligned}$$

При выполнении частичного склеивания может оказаться, что грамматике G_{n+1} одновременно принадлежат такие правила с одинаковыми правыми частями

$$A_{i_1 \dots i_t} \to \xi \ , \tag{1.3}$$

$$A_{n_1 \dots n_k} \to \xi \ , \tag{1.4}$$

что $\{i_1, i_2, ...i_t\} \subset \{n_1, n_2, ...n_k\}$. Назовем обобщенным склеиванием по индексам такое построение грамматики G_{n+1} по грамматике G_n, когда

1) выполнено частичное склеивание G_n к G_{n+1};

2) если среди множества полученных правил грамматики G_{n+1} окажутся правила вида (1.3), (1.4), то из множества правил грамматики удаляется правило (1.3) как более слабое.

В нашем примере грамматике одновременно принадлежат правила $A_1 \to aaA_{12}$, $A_2 \to aaA_{12}$, $A_{12} \to aaA_{12}$, первые два из которых можно удалить. Правые части правил грамматики, полученной в результате такого удаления, различны:

$$\begin{aligned} G_2: \quad & S \to bA_1|baA_2|bA_{12}|baA_{12} \\ & A_1 \to aaA_1|d \\ & A_2 \to aaA_2|c \\ & A_{12} \to ab|aaA_{12} \end{aligned}$$

Можно доказать следующую теорему.

Теорема 1.6. В результате последовательного применения к любой КС–грамматике операции обобщенного склеивания строится эквивалентная КС–грамматика, все правила которой имеют различные правые части.

Доказательство. Очевидно, что последовательное применение алгоритма обобщенного склеивания преобразует любую КС-грамматику к такой форме, когда все правила имеют различные правые части. Осталось показать эквивалентность исходной грамматики и грамматики, полученной в результате операции обобщенного склеивания. Сначала заметим, что частичное склеивание является эквивалентным преобразованием грамматики G_n, т.к. замена (1.1) и подстановка (1.2) каждому выводу в G_n взаимно однозначно ставят вывод в G^1_{n+1}, полученной из G_n операцией частичного склеивания. Пусть грамматика G_{n+1} получена из грамматики G_n операцией обобщенного склеивания. Покажем, что удаление более слабых правил (1.3) при наличии правил (1.4) не нарушает эквивалентности грамматик. Действительно, пусть в выводе некоторой цепочки x в грамматике G_n применялось правило вида (1.3):

$$S \overset{*}{\Rightarrow} z_1 A_{i_1,\dots,i_t} z_2 \to z_1 \xi z_2 \overset{*}{\Rightarrow} x.$$

Нетерминал $A_{i_1,\dots,i_t}$ появился в этом выводе на одном из предшествуюших шагов:

$$S \overset{*}{\Rightarrow} y_1 B y_2 \Rightarrow y_1 \omega A_{i_1 \dots i_t} \eta y_2 = z_1 A_{i_1 \dots i_t} z_2 \to z_1 \xi z_2 \overset{*}{\Rightarrow} x.$$

Но тогда по правилу подстановки при выполнении операции частичного склеивания в множество правил новой грамматики G_{n+1} включается новое првило

$$B \to \omega A_{n_1 n_2 \ldots n_t} \eta$$

и, следовательно, существует эквивалентный вывод в G_{n+1}:

$$S \overset{*}{\Rightarrow} y_1 B y_2 \Rightarrow y_1 \omega A_{n_1 n_2 \ldots n_t} \eta y_2 = z_1 A_{n_1 n_2 \ldots n_t} z_2 \to z_1 \xi z_2 \overset{*}{\Rightarrow} x.$$

Таким образом, $L(G_n) = L(G_{n+1})$. $\square$

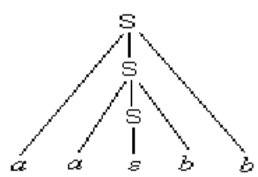
1.5.3 Неукорачивающие грамматики

Определение 1.17. Грамматика, не содержащая правил с пустой правой частью, называется неукорачивающей грамматикой.

При выводе в неукорачивающей грамматике длина выводимой цепочки не уменьшается при переходе от k–го шага вывода к $(k+1)$–му. В грамматике с правилами вида $A \to \varepsilon$ длина выводимой цепочки может уменьшаться:

$$\begin{aligned} G: \quad & S \to aSb|\varepsilon \\ & S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb. \end{aligned}$$

В соответствии с определением (1.17) грамматика называется укорачивающей, если на некотором шаге может уменьшиться длина разбора выводимой цепочки. Восходящий разбор укорачивающих грамматик, как правило, сложнее по сравнению с разбором в неукорачивающих грамматиках, так как необходимо найти такой фрагмент входной цепочки, где можно вставить пустую цепочку ε:



Основным требованием, предъявляемым к языку программирования, является возможность определить для каждой данной последовательности символов, является ли она программой, написанной на этом языке. Это необходимо в первую очередь для того, чтобы сделать возможной автоматическую трансляцию программ. На формальном уровне это означает, что должна быть разрешима проблема принадлежности любой цепочки любому заданному КС–языку. Другими словами, должен существовать алгоритм, позволяющий определить, принадлежит ли произвольная заданная цепочка x произвольному заданному КС–языку L. С этой точки зрения особенный интерес представляют неукорачивающие КС–грамматики в силу следующих теорем.

Теорема 1.7. Пусть $G = (V_T, V_N, P, S)$ есть неукорачивающая КС–грамматика и n — число нетерминальных символов в множестве V_N. Цепочка $x \in V_T^*$ длины k тогда и только тогда принадлежит языку $L(G)$, когда существует ее вывод

$$S \Rightarrow U_1 \Rightarrow \ldots \Rightarrow U_p = x, \tag{1.5}$$

такой, что $p \leq n^k$.

Доказательство. Пусть $G = (V_T, V_N, P, S)$ и $x \in V_T^*$. Согласно определению языка $L(G)$, если существует вывод (1.5), то $x \in L(G)$. Допустим теперь, что цепочка x выводима из S в G. Пусть r — наименьшее число, для которого существует вывод $S \Rightarrow Z_1 \Rightarrow \ldots \Rightarrow Z_r = x$. Поскольку G — неукорачивающая КС–грамматика, то $|Z_i| \leq |Z_{i+1}|$ для каждого i. Предположим, что $r > n^k$. Тогда в выводе x найдутся цепочки $Z_l = Z_m$ при $l < m$. Отсюда следует, что имеется более короткий вывод $S \Rightarrow Z_1 \Rightarrow \ldots \Rightarrow Z_l \Rightarrow Z_{m+1} \Rightarrow \ldots \Rightarrow Z_r = x$, что противоречит свойству минимальности рассмотренного нами вывода.□

Теорема 1.8. Существует алгоритм, позволяющий узнать, принадлежит ли произвольная цепочка языку, порождаемому данной неукорачивающей КС–грамматикой.

Доказательство. Пусть x — произвольная цепочка из Σ^* и G — некоторая неукорачивающая КС–грамматика с тем же множеством терминальных символов. Пусть длина цепочки x равна k. Из теоремы 1.7 следует, что $x \in L(G)$ тогда и только тогда, когда x порождается с помощью некоторого вывода, длина которого не превышает $m = n^{|x|}$, где n — число нетерминальных символов в G. Тогда достаточно просмотреть все выводы, длины которых не превышают m. Цепочка x принадлежит $L(G)$ тогда и только тогда, когда ее можно получить с помощью хотя бы одного из рассмотренных выводов.□

Представляет интерес преобразование любой КС–грамматики в эквивалентную неукорачивающую грамматику. Если G — КС–грамматика и $\varepsilon \in L(G)$, то хотя бы одно правило вида $A \rightarrow \varepsilon$ в множестве правил G имеется. Если $\varepsilon \notin L(G)$, то по грамматике G всегда можно построить КС–грамматику, тоже порождающую язык $L(G)$, но не содержащую ни одного правила с пустой правой частью. Алгоритм такого построения мы рассмотрим при доказательстве следующей теоремы.

Теорема 1.9. Для произвольной КС–грамматики, порождающей язык без пустой цепочки, можно построить эквивалентную неукорачивающую КС–грамматику.

Доказательство. Построим множество всех нетерминальных символов грамматики $G = (V_T, V_N, P, S)$, из которых выводится пустая цепочка. С этой целью выделим следующие множества нетерминалов:

$$
\begin{array}{l}
W_1 = \{A | A \rightarrow \varepsilon \in P\}, \\
W_{m+1} = W_m \cup \{B | B \rightarrow \phi \in P, \phi \in W_m^*\}.
\end{array}
$$

Ясно, что $A \stackrel{*}{\Rightarrow} \varepsilon$ тогда и только тогда, когда $A \in W_n$ для некоторого n ($n < |V_N|$).

Рассмотрим множество P_1 , состоящее из всех правил $A \rightarrow \tilde{\alpha}$, полученных из правил $A \rightarrow \alpha \in P$ при удалении из α некоторых (возможно, никаких) вхождений символов из множества W_n:

$$
\begin{array}{l}
P_1^0 = P \setminus \{A \rightarrow \varepsilon | A \rightarrow \varepsilon \in P\} \\
P_1^{i+1} = \{A \rightarrow \tilde{\alpha} | A \rightarrow \alpha \in P_1^i;\ \alpha = \alpha_1 B \alpha_2;\ B \in W_n \& \tilde{\alpha} = \alpha_1 \alpha_2\} .
\end{array}
$$

Грамматика $G_1 = (V_T, V_N, P_1, S)$ является неукорачивающей. Покажем, что она эквивалентна G. Пусть $x \in L(G_1)$, тогда существует вывод x в G_1 . Рассмотрим первый левый шаг вывода, на котором применялось первое правило, не принадлежащее P. По построению множеству P принадлежит такое правило $A \rightarrow \alpha$, что $\alpha = x_1 A_1 x_2 A_2 \ldots x_r A_r x_{r+1}$ и $A_1, A_2, \ldots A_r \in Wn$. Это означает, что в G выводимо $A_1 \stackrel{*}{\Rightarrow} \varepsilon, \ldots, A_r \stackrel{*}{\Rightarrow} \varepsilon$, но тогда рассмотренный шаг вывода в G_1 можно заменить на вывод в G :

$$
x_1 A_1 x_2 A_2 ... x_r A_r x_{r+1} \Rightarrow x_1 x_2 A_2 ... x_r A_r x_{r+1} \Rightarrow \ldots \Rightarrow x_1 x_2 ... x_{r+1} .
$$

Выполняя такие преобразования на каждом шаге вывода в G_1 , получим вывод x в G, следовательно, $L(G_1) \subseteq L(G)$. Аналогично можно показать, что $L(G) \subseteq L(G_1)$. С этой целью вывод x в G преобразуется в вывод x в G_1 удалением справа налево всех правил вида $A \to \varepsilon$.□

Пример 1.7. Пусть задана КС–грамматика

$$\begin{aligned} G: \quad & S \to AbA|cA|Bbb \\ & A \to aAb|\varepsilon \\ & B \to AA|ca. \end{aligned}$$

Построим множества нетерминалов, из которых выводится ε :

1) $W_1 = \{A\}$, т.к. имеется правило $A \to \varepsilon$;
2) $W_2 = \{A, B\}$, т.к. имеется правило $B \to AA$ и $A \in W_1$;
3) $W_3 = W_2$.

Построим неукорачивающую грамматику, эквивалентную исходной:

$$\begin{aligned} G: \quad & S \to AbA|cA|Bbb|Ab|bA|b|c|bb \\ & A \to aAb|ab \\ & B \to AA|ca|A. \end{aligned}$$

Определение 1.18. КС–грамматики G_1 и G_2 эквивалентны с точностью до пустой цепочки, если

$$L(G_1) \setminus \{\varepsilon\} = L(G_2) \setminus \{\varepsilon\}.$$

Следовательно, для любой КС–грамматики можно построить неукорачивающую, эквивалентную исходной с точностью до ε.

1.5.4 Непродуктивные нетерминалы

Рассмотрим грамматику

$$\begin{aligned} G: \quad & S \to bS|a|aA \\ & A \to aAb. \end{aligned}$$

Нетерминальный символ A не участвует в выводе терминальных цепочек и поэтому может быть исключен из грамматики вместе со всеми правилами, в которые он входит.

Определение 1.19. Нетерминальный символ A называется непродуктивным (непроизводящим), если он не порождает ни одной терминальной цепочки, т.е. не существует вывода $A \stackrel{*}{\Rightarrow} x$, где $x \in V_T^*$.

Представляет интерес удаление из грамматики всех непродуктивных нетерминальных символов.

Теорема 1.10. Для произвольной КС–грамматики можно построить эквивалентную КС–грамматику, все нетерминальные символы которой продуктивны.

Доказательство. Пусть $G = (V_T, V_N, P, S)$ — произвольная КС–грамматика. Построим множество всех продуктивных нетерминалов этой грамматики. С этой целью выделим множество W_1 нетерминалов, стоящих в левой части терминальных правил:

$$W_1 = \{A|A \to \phi \in P; \phi \in V_T^*\}.$$

Затем построим множества $W_2, W_3, \ldots, W_n$ (n — число нетерминальных символов в G):

$$W_{k+1} = W_k \cup \{B|B \to x \in P;\ x \in (V_T \cup W_k)^*\}.$$

Все $B \in W_1$ продуктивны по построению W_1. Пусть продуктивны все $B \in W_k (k \geq 1)$. Покажем, что продуктивными являются и все $B \in W_{k+1}$. Действительно, если $B \in W_{k+1}$, то либо $B \in W_k$ и является продуктивным, либо $B \notin W_k$, но тогда $B \to x \in P$ и $x \in (V_T \cup W_k)^*$. Пусть $x = x_1 A_1 x_2 \dots x_m A_m x_{m+1}$, тогда все A_i являются продуктивными, следовательно, B — продуктивный нетерминал. Пусть теперь B — продуктивный нетерминал, минимальный терминальный вывод из B имеет длину $k+1$ и $B \Rightarrow x_1 A_1 x_2 \dots x_m A_m x_{m+1} \stackrel{*}{\Rightarrow} y \in V_T^*$, все $x \in V_T^*$, все $A_i \in V_N^*$. Для каждого A_i существует вывод $A_i \stackrel{*}{\Rightarrow} y_i V_T t^*$ длины не более k, тогда все $A_i \in W_i (i \leq k)$ и по построению множества W_{k+1} ему принадлежит B. Таким образом, показали, что $B \in W_n$ тогда и только тогда, когда он продуктивен. Все символы множества $V_N \setminus W_n$ являются непродуктивными, не используются в выводе никакой терминальной цепочки и их можно удалить из грамматики вместе со всеми правилами, в которые они входят.$\square$

Пример 1.8. Пусть задана грамматика:

$$\begin{aligned} G: \quad & S \to SA|BSb|bAb \\ & A \to aSa|bb \\ & B \to bBb|BaA. \end{aligned}$$

Построим множество продуктивных нетерминалов:

1) $W_1 = \{A\}$, т.к. имеется правило $A \to bb$;

2) $W_2 = \{A, S\}$, т.к. имеется правило $S \to bAb$ и $A \in W_1$;

3) $W_3 = W_2$.

Эквивалентная исходной грамматика, все символы которой продуктивны, имеет вид:

$$\begin{aligned} G_1: \quad & S \to SA|bAb \\ & A \to aAa|bb. \end{aligned}$$

1.5.5 Независимые нетерминалы

Существует еще один тип нетерминальных символов, которые можно удалять из грамматики вместе с правилами, в которые они входят. Рассмотрим, например, грамматику

$$\begin{aligned} G: \quad & S \to aS|b \\ & A \to aAb|ab. \end{aligned}$$

Нетерминальный символ A не участвует ни в каком выводе в этой грамматике, т.к. из аксиомы нельзя вывести цепочку, содержащую A.

Определение 1.20. В КС–грамматике G нетерминальный символ A зависит от нетерминального символа B, если в G существует вывод $A \stackrel{*}{\Rightarrow} xBy$.

В рассмотренном выше примере аксиома S не зависит от символа A, поэтому A можно удалить из грамматики.

Теорема 1.11. Для произвольной КС–грамматики можно построить эквивалентную КС–грамматику, аксиома которой зависит от всех нетерминальных символов.

Доказательство. Пусть $G = (V_T, V_N, P, S)$ — произвольная КС–грамматика. Построим множество нетерминалов, от которых зависит аксиома. С этой целью выделим множества $W_1, W_2, ...W_n$:

$$\begin{aligned} & W_1 = \{S\}, \\ & W_{k+1} = W_k \cup \{B | A \to xBy \in P, A \in W_k\}. \end{aligned}$$

Аналогично доказательству теоремы 1.10 можно показать, что $B \in W_n$ тогда и только тогда, когда S зависит от B (доказательство оставляется в качестве упражнения).

Все нетерминалы, не содержащиеся в W_n, можно удалить из грамматики вместе с правилами, в которые они входят. $\square$

Пример 1.9. Пусть задана КС–грамматика

$$\begin{aligned} G:\ & S \to AS|bb \\ & A \to aAb|ab \\ & B \to SB|aAb. \end{aligned}$$

Найдем нетерминалы, от которых зависит аксиома:

1) $W_1 = \{S\}$;

2) $W_2 = \{S, A\}$, т.к. имеется правило $S \to AS$ и $S \in W_1$;

3) $W_3 = W_2$.

Эквивалентная грамматика, аксиома которой зависит от всех нетерминальных символов:

$$\begin{aligned} G_1:\ & S \to AS|Sb \\ & A \to aAb|ab \end{aligned}$$

Определение 1.21. КС–грамматика $G = (V_T, V_N, P, S)$ называется приведенной, если S зависит от всех нетерминалов из V_N и в V_N нет непродуктивных символов.

Из теорем 1.10 и 1.11 следует, что для любой КС–грамматики можно построить приведенную эквивалентную КС–грамматику.

1.5.6 Терминальные правила

Рассмотрим в КС–грамматике $G = (V_T, V_N, P, S)$ удаление правила с терминальной правой частью $A \to \beta$, где $\beta \in V_T^*$. Любой вывод с использованием этого правила имеет вид

$$S \Rightarrow^* x_1 A x_2 \Rightarrow x_1 \beta x_2.$$

Но нетерминал A в сентенциальной форме $x_1 A x_2$ появился на некотором предыдущем шаге вывода $B \to uAv$, тогда

$$S \overset{*}{\Rightarrow} z_1 B z_2 \overset{*}{\Rightarrow} z_1 uAv z_2 = x_1 A x_2 \Rightarrow x_1 \beta x_2\ .$$

Если в правило грамматики $B \to uAv$ вместо A подставить β, получим правило $B \to u\beta v$ и длина вывода сократится на один шаг при неизменности выводимой цепочки. Следовательно, для того, чтобы удалить терминальное правило грамматики $A \to \beta$, необходимо рассмотреть следующие варианты :

а) для A больше нет правил, тогда во всех правых частях A заменяется на β;

б) для A есть другие правила, тогда добавляются новые правила, в которых A заменяется на β.

Пример 1.10. Рассмотрим грамматику, порождающую идентификаторы (см. 1.2), обозначив для наглядности символами b — букву, c — цифру:

$$\begin{aligned} G:\ & S \to bA \\ & A \to AZ|\varepsilon \\ & Z \to b|c. \end{aligned}$$

Удалим правила для Z, получим эквивалентную грамматику

$$\begin{aligned} G_1:\ & S \to bA \\ & A \to Ab|Ac|\varepsilon. \end{aligned}$$

1.5.7 Леворекурсивные и праворекурсивные правила

Некоторые специальные методы грамматического разбора неприменимы к леворекурсивным или праворекурсивным грамматикам, поэтому рассмотрим устранение левой или правой рекурсии. Следует отметить, что бесконечный язык порождается грамматикой с конечным числом правил только благодаря рекурсии, следовательно, вообще избавиться от рекурсии в правилах невозможно. Можно лишь преобразовать один вид рекурсии в другой.

Теорема 1.12. Для любой леворекурсивной КС–грамматики существует эквивалентная праворекурсивная.

Доказательство. Пусть нетерминал A леворекурсивен, т.е. правила для него имеют вид

$$A \to Ax_1|Ax_2|\dots|Ax_p|w_1|w_2|\dots|w_r, \tag{1.6}$$

где x_i и w_j — цепочки над $V_T \cup V_N$. Введем дополнительные нетерминалы B и D и заменим (1.6) на эквивалентные правила

$$\begin{array}{ll} A \to & AB|D \\ B \to & x_1|x_2|\dots|x_p \\ D \to & w_1|w_2|\dots|w_r. \end{array}$$

Вывод из A имеет вид $A \Rightarrow AB \Rightarrow ABB \Rightarrow ... \Rightarrow AB^* \Rightarrow DB^*$ и, следовательно, для A можно определить эквивалентные правила

$$\begin{array}{ll} A \to & DK \\ K \to & BK|\varepsilon. \end{array}$$

Выполняя подстановку B и D в эти правила, получим праворекурсивные правила

$$\begin{array}{l} A \to w_1K|w_2K|...|w_rK \\ K \to x_1K|x_2K|...|x_pK|\varepsilon\ . \end{array}$$

□

Пример 1.11. Для грамматики

$$\begin{array}{ll} G: & S \to SaA|ab \\ & A \to Aab|aAb|c \end{array}$$

можно построить эквивалентную нелеворекурсивную

$$\begin{array}{ll} G_1: & S \to abT \\ & T \to aAT|\varepsilon \\ & A \to aAbR|cR \\ & R \to abR|\varepsilon. \end{array}$$

Аналогично можно построить алгоритм устранения правой рекурсии и доказать эквивалентность соответствующего преобразования.

Теорема 1.13. Для любой праворекурсивной КС–грамматики существует эквивалентная леворекурсивная.

Доказательство. Пусть нетерминал A праворекурсивен, т.е. правила для него имеют вид

$$A \to x_1A|x_2A|\dots|x_pA|w_1|w_2|\dots|w_r, \tag{1.7}$$

где x_i и w_j — цепочки над $V_T \cup V_N$. Введем дополнительные нетерминалы B и D и заменим (1.7) на эквивалентные правила

$$\begin{array}{l} A \to BA|D \\ B \to x_1|x_2|\dots|x_p \\ D \to w_1|w_2|\dots|w_r. \end{array}$$

Вывод из A имеет вид $A \Rightarrow BA \Rightarrow BBA \Rightarrow ... \Rightarrow B^*A \Rightarrow B^*D$ и, следовательно, для A можно определить эквивалентные правила

$$\begin{array}{l} A \to KD \\ K \to KB|\varepsilon. \end{array}$$

Выполняя подстановку B и D в эти правила, получим леворекурсивные правила

$$\begin{array}{l} A \to Kw_1|Kw_2|...|Kw_r \\ K \to Kx_1|Kx_2|...|Kx_p|\varepsilon. \end{array}$$

□

1.6 Теорема о языке $a^n b^n c^n$

В 1.2 мы сформулировали, но не доказали теорему о языке $a^n b^n c^n$. Рассмотрим теперь ее доказательство. Сразу отметим, что при доказательстве этой теоремы мы будем использовать только понятие выводимости и определение дерева грамматического разбора, не пытаясь применить полученные из теоремы 1.2 выводы о незамкнутости семейства КС–языков относительно операций пересечения, дополнения и разности.

Лемма 1.1. Для любой КС – грамматики, порождающей бесконечный язык, существуют такие натуральные числа p и q, что каждая цепочка $w \in L(G), |w| > p$, может быть представлена в виде $w = xuyvz$, где $|uv| > q$ и для любого $n > 0$ цепочка $xu^n yv^n z \in L(G)$.

Доказательство. Рассмотрим всевозможные выводы $S \overset{*}{\Rightarrow} t$ терминальных цепочек $t \in V_T^*$ из аксиомы S, удовлетворяющие тем условиям, что в дереве разбора цепочки t на пути из корня в любой лист один и тот же нетерминал не встречается два раза. Возьмем в качестве числа p максимальную длину таких цепочек t. Возьмем произвольную цепочку w такую, что $|w| > p$ и $S \overset{*}{\Rightarrow} w$. По построению числа p в дереве вывода цепочки w на каком–то пути из S в лист существует повторяющийся нетерминал A_i (см. рис. 1.6).

Мы знаем, что любая грамматика может быть преобразована так, что она является неукорачивающей и не содержит правил вида нетерминал–нетерминал, следовательно, $|uv| > 0$. Возьмем в качестве числа $q = |uv|$. Тогда существует вывод $A_i \overset{*}{\Rightarrow} u^n A_i v^n$ для любого n. Следовательно, в G существует вывод

$$S \overset{*}{\Rightarrow} xA_iz \overset{*}{\Rightarrow} xu^n A_i v^n z \Rightarrow xu^n yv^n z.$$

□

Теорема 1.2. Язык $a^n b^n c^n$, $n \geq 0$ — не КС–язык.

Доказательство. Пусть $a^n b^n c^n$ — КС–язык, т.е. существует КС–грамматика G, его порождающая. По доказанной лемме существует такая цепочка $xvyuz \in L(G)$, что $xv^k yu^k z \in L(G)$. Рассмотрим все варианты определения подцепочки v в слове

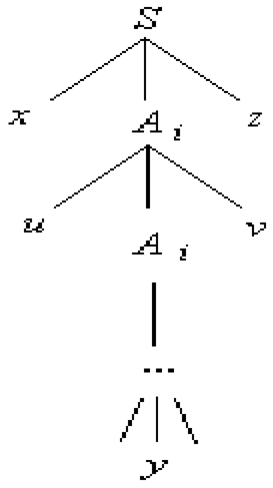


Рис. 1.6: Повторяющийся нетерминал A_i в дереве вывода.

$a^n b^n c^n$: a^l, $a^l b^m$, b^l, $b^l c^m$, c^m, $a^l b^m c^p$. Покажем, что ни один из этих вариантов определения v не может иметь места.

Пусть $v = a^l$, тогда $a^t (a^l)^k y u^k z \in L(G)$, что невозможно из–за нарушения соотношения между равным числом символов a и b или a и c.

Пусть $v = a^l b^m$, тогда $a^t (a^l b^m)^k y u^k z L(G)$, чего быть не может, т.к. при $k > 1$ в полученной цепочке после символа b встречается символ a.

Аналогично можно показать невозможность оставшихся вариантов определения v. Следовательно, язык $a^n b^n c^n$ не может порождаться КС–грамматикой. Пример грамматики типа 0, порождающей этот язык, приведен в примере 1.2. □

1.7 Контрольные вопросы к разделу

1. Поясните понятие терминальных символов грамматики.

2. Чем терминальные символы отличаются от нетерминальных?

3. Чем отличаются КС–грамматики от грамматик непосредственно составляющих?

4. Какой класс грамматик является более широким: грамматики типа 0 или грамматики типа 2?

5. Как построить грамматику для итерации языка, заданного КС–грамматикой?

6. Как построить грамматику для объединения языков, заданных КС–грамматиками?

7. Замкнуто ли множество КС–языков относительно операции пересечения?

8. Можно ли для произвольной КС–грамматики построить эквивалентную неукорачивающую? Если можно, то как это сделать?

9. Как в заданной КС–грамматике избавиться от правил с одинаковыми правыми частями?

10. Чем характеризуется восходящая стратегия разбора?

11. Приведите пример КС–грамматики, для которой возникают затруднения при нисходящем грамматическом разборе.

12. Какая КС-грамматика называется приведенной?
13. К какому классу языков относится язык $a^n b^n c^n$?
14. Что называется деревом грамматического разбора?
15. Как устранить левую рекурсию в правилах КС–грамматики?
16. Можно ли устранить центральную рекурсию в правилах КС–грамматики?
17. Устраните правую рекурсию в грамматике

$$G : S \to SaaSbS|SbS|bSbb|aaSb.$$

18. Устраните левую рекурсию в грамматике задания 17.
19. Какую цель преследуют при устранении правил вида "нетерминал — нетерминал"?
20. Приведите пример грамматики, в которой имеет смысл устранить некоторые правила с терминальной правой частью.

1.8 Упражнения к разделу

Задание. Построить КС-грамматику, порождающую заданный язык и привести пример дерева разбора в построенной грамматике.

1.8.1 Задача

Дан язык

$$(ab)^{n+1}c^{3n} \cup (a^*bc^+ \cup (ad)^+)^*, n \geq 0.$$

Построить КС–грамматику и привести пример дерева разбора в построенной грамматике.

Решение. Обозначим символом S аксиому грамматики. Язык $(ab)^{n+1}c^{3n} \cup (a^*bc^+ \cup (ad)^+)^*$ построен с помощью операции объединения языков $(ab)^{n+1}c^{3n}$ и $(a^*bc^+ \cup (ad)^+)^*$, поэтому для аксиомы S необходимо определить два правила

$$S \longrightarrow A|B,$$

где символы A и B соответствуют указанным языкам. Рассмотрим язык $(ab)^{n+1}c^{3n}$, который должен выводиться из нетерминального символа A. Для того, чтобы получить правила для нетерминала A, запишем равенство $(ab)^{n+1}c^{3n} = (ab)^n ab(ccc)^n$ и воспользуемся правилом самовставленя нетерминального символа. Получаем правила грамматики:

$$A \longrightarrow abAccc|ab.$$

Язык $(a^*bc^+ \cup (ad)^+)^*$, соответствующий нетерминалу B, получен с помощью операции итерации языка $a^*bc^+ \cup (ad)^+$. Поэтому записываем правила граммматики:

$$B \longrightarrow BT|\varepsilon$$

$$T \longrightarrow R|F$$

Языки, соответствующие нетерминалам R и F, получаются с помощью операций конкатенации, итерации и усеченной итерации над элементарными цепочками. Таким образом, получаем КС-грамматику

$$
\begin{aligned}
G:\quad & S \longrightarrow A|B \\
& A \longrightarrow abAccc|ab \\
& B \longrightarrow BT|\varepsilon \\
& T \longrightarrow R|F \\
& R \longrightarrow MbK \\
& M \longrightarrow Ma|\varepsilon \\
& K \longrightarrow Kc|c \\
& F \longrightarrow Fad|\varepsilon.
\end{aligned}
$$

Можно несколько упростить грамматику, удалив нетерминал R:

$$
\begin{aligned}
G:\quad & S \longrightarrow A|B \\
& A \longrightarrow abAccc|ab \\
& B \longrightarrow BT|\varepsilon \\
& T \longrightarrow MbK|F \\
& M \longrightarrow Ma|\varepsilon \\
& K \longrightarrow Kc|c \\
& F \longrightarrow adF|\varepsilon.
\end{aligned}
$$

Построим дерево разбора цепочки $ababccc$. Эта цепочка получена с помощью вывода

$$S \Rightarrow A \Rightarrow abAccc \Rightarrow ababccc,$$

поэтому дерево грамматического разбора этой цепочки имеет вид, представленный на рис. 7.7.

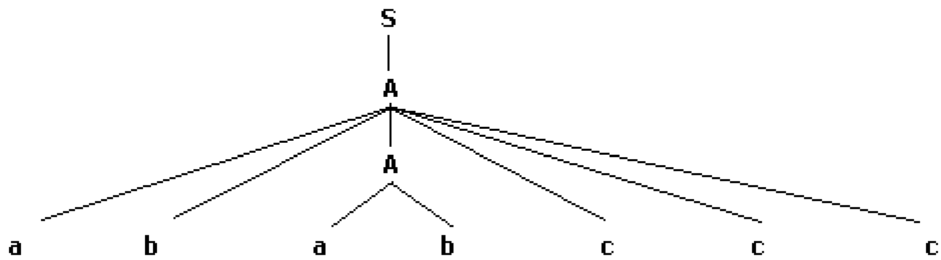


Рис. 1.7: Дерево разбора, соответствующее выводу $S \Rightarrow A \Rightarrow abAccc \Rightarrow ababccc$

Построим теперь дерево разбора цепочки $aabcadadad$. Эта цепочка может быть получена с помощью различных выводов:

$$
\begin{gathered}
S \Rightarrow B \Rightarrow BT \Rightarrow BTT \Rightarrow BTTT \Rightarrow BTTTT \Rightarrow TTTT \Rightarrow \\
\Rightarrow MbKTTT \Rightarrow MabKTTT \Rightarrow MaabKTTT \Rightarrow aabKTTT \Rightarrow \\
\Rightarrow aabcTTT \Rightarrow aabcFTT \Rightarrow aabcadFTT \Rightarrow aabcadTT \Rightarrow aabcadFT \Rightarrow \\
\Rightarrow aabcadadFT \Rightarrow aabcadadT \Rightarrow aabcadadF \Rightarrow \\
\Rightarrow aabcadadadF \Rightarrow aabcadadad. \qquad (7.7)
\end{gathered}
$$

или

$$
\begin{gathered}
S \Rightarrow B \Rightarrow BT \Rightarrow BTT \Rightarrow TT \Rightarrow MbKT \Rightarrow \\
\Rightarrow MabKTT \Rightarrow MaabKT \Rightarrow aabKT \Rightarrow aabcT \Rightarrow aabcF \Rightarrow \\
\Rightarrow aabcadF \Rightarrow aabcadadF \Rightarrow aabcadadadF \Rightarrow aabcadadad. \qquad (7.8)
\end{gathered}
$$

Очевидно, что грамматика, которую мы построили, является неоднозначной. Соответствующие этим выводам деревья разбора представлены на рис. 7.8 и 7.9.

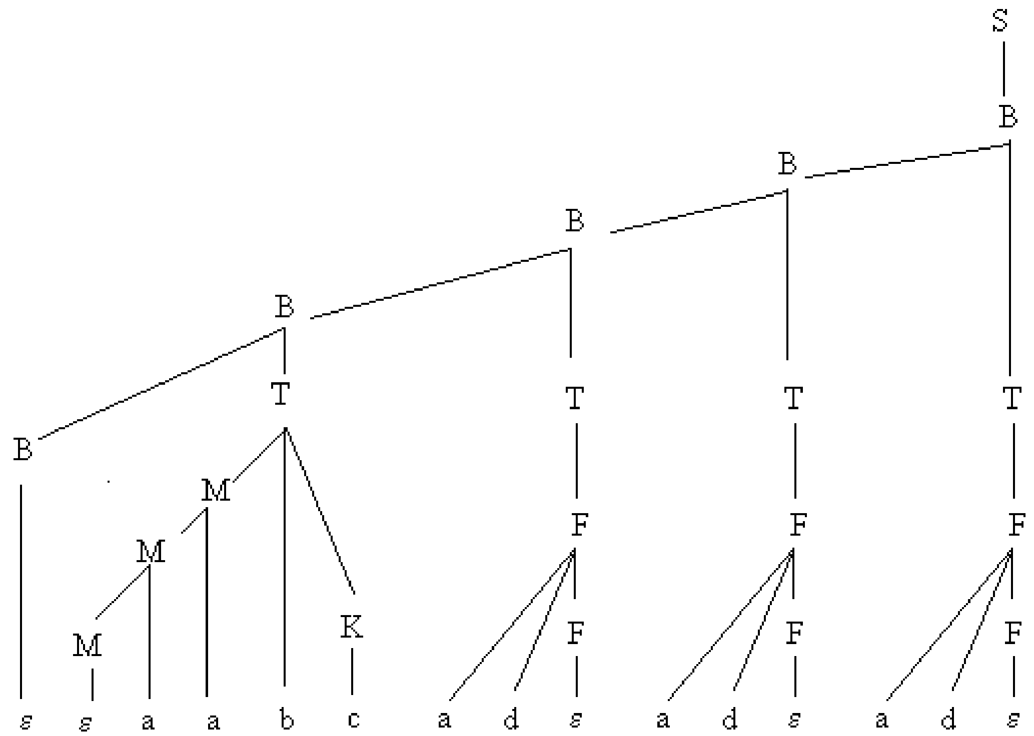


Рис. 1.8: Дерево разбора для вывода (7.7).

1.8.2 Варианты заданий

1. $(abc^+ \cup a^n cccb^{2n+1} \cup (aba)^+)^*, n \geq 0$
2. $((acb)^* ad^+ (a^+ba)^{n+1} cb^{3n+1} \cup (a^*bc^+)^*)^* \cup a^*b^+, n \geq 0$
3. $ac^*ad^+(a^+ba)^{2n+1}cb^n \cup (ba^*c^*b)^* \cup (ab)^{2k}b^k, n, k \geq 0$
4. $c^+d^+(a^*ba)^{n+1}cb^{4n} \cup ((ba^*c^*b)^+ \cup (ca^+b)^{2k}b^k)^*, n \geq 0, k \geq 1$
5. $(cca^{2n+2}ab^+(bc)^n)^* \cup (a^*b^+ \cup (bc)^k dda^{2k})^+, n, k \geq 1$
6. $ac^*ad^+(a^+ba)^{2n+1}cb^n \cup (ba^*c^*b)^* \cup (ab)^{2k}b^k, n, k \geq 1$
7. $(bc^* \cup (ad)^n cb^{2n+1} \cup (ba^*)^+)^*, n \geq 0$
8. $(a^*ba)^{n+1}cb^{2n} \cup ((ba^*c^*b)^+ \cup (ca^*b)^{3k}ccb^k)^*, n \geq 1, k \geq 0$
9. $(ac^*ad^+(a^+ba)^{2n+1}cb^{3n} \cup (a^*bc^+)^*)^* \cup a^kb^k, n, k \geq 0$
10. $ac^+ad^+(a^+ba)^{2n+1}cb^n \cup (ba^*c^*b)^+ \cup (ca^+b)^{2k}b^k, n \geq 0, k \geq 1$
11. $(a^{2n+2}b^+(bc)^n \cup (a^*b^+ \cup a + c^* \cup (bc)^k dda^{2k})^+)^*, n, k \geq 0$
12. $(bc^*ad \cup (a^+ba)^n cb^{3n+2} \cup (ba^*)^*)^*, n \geq 0$
13. $(a^{2n+2}b^+(bc)^n \cup (a^mb^m \cup c + b^* \cup (aac)^{k+2}dda^{2k})^+)^*, n, k, m \geq 0$
14. $(ac^*ad^+(a^+ba)^{2n+1}cb^n \cup (a^*bc^+)^*)^+ \cup a^{2k}b^k, n, k \geq 0$
15. $(a^{2n+2}b^+(bc)^n(a^*b^* \cup c + b^* \cup (aac)^{k+2}dda^{2k})^+)^*, n, k \geq 0$
16. $(ab^{n+2}c(bc)^n)^* \cup (a^+b^+ \cup abc^k dda^{2k})^+, n, k \geq 1$
17. $ac^+ad^+(a^+ba)^{2n+1}cb^n \cup ((ba^*c^*b)^+ \cup (ca^+b)^{2k}b^k)^*, n \geq 0, k \geq 1$
18. $(a^{2n+2}b^+(bc)^n)^+ \cup (a^*b^+ \cup a + c^* \cup (bc)^k dda^{2k})^+, n, k \geq 1$
19. $((acb)^*ad^+ \cup (a^+ba)^{n+1}cb^{3n+1} \cup (a^*bc^+)^*)^*, n \geq 0$
20. $(ac^*ad^+(a^+ba)^{2n+1}cb^n)^+ \cup (a^*bc^+)^* \cup a^{2k}b^k, n, k \geq 0$

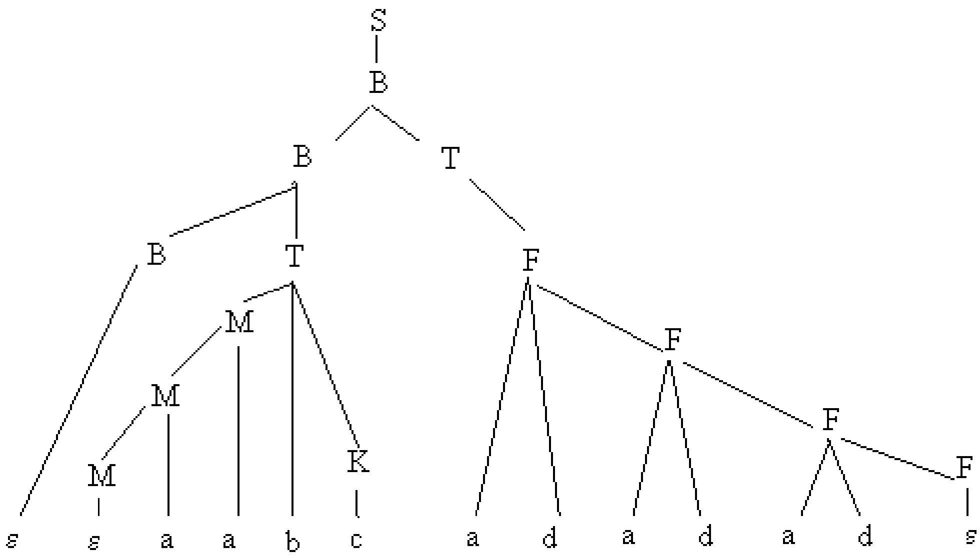


Рис. 1.9: Дерево разбора для вывода (7.8).

1.9 Тесты для самоконтроля к разделу

1. Какой язык порождает грамматика

$$G: \quad \begin{array}{l} S \longrightarrow ASa|b \\ A \longrightarrow aA|c \end{array}$$

Варианты ответов:

а) $(a^*c)^n ba^n, n \geq 0$;
б) $(a^*c)^n ba^n, n > 0$;
в) $(a^*c)^* ba^*$;
г) $a^*c^*b^*a^*$;
д) $a^n c^n b^n a^n, n \geq 0$;
е) $a^n c^n b^n a^n, n > 0$;
ж) $a^n cbca^n, n > 0$;
ж) $a^n cbca^n, n \geq 0$;

Правильный ответ: а.

2. Какой из перечисленных ниже языков не является регулярным:

а) $(ac)^n ba^m,\ n, m > 0$;
б) $(a^*c)^n ba^n,\ n > 0$;
в) $(a^*c)^* ba^*$;
г) $a^*c^*b^*a^*$;
д) $a^n cb^m a^k,\ n, m, k > 0$;
ж) $a^n cbca^+,\ n > 0$;
ж) $a^* cbca^n,\ n \geq 0$;

Правильный ответ: б.

3. Какие символы являются продуктивными в грамматике

$$\begin{aligned} G: \quad & S \longrightarrow ASa|bD|F \\ & A \longrightarrow aA|cAS|DaF \\ & B \longrightarrow aA|\varepsilon \\ & D \longrightarrow aA|cB \\ & F \longrightarrow aA|cF|Fa \end{aligned}$$

Варианты ответов:

а) все нетерминалы S, A, B, D, F продуктивны;

б) S;

в) S, B, D;

г) S, A, D, F;

д) A, B, D, F;

е) B, D;

ж) все нетерминалы непродуктивны; .

Правильный ответ: в.

4. Дана КС–грамматика:

$$\begin{aligned} G_3: \quad & S \longrightarrow aSA|Ab|Bc \\ & A \longrightarrow aAb|\varepsilon \\ & B \longrightarrow AA|aBc. \end{aligned}$$

Какая неукорачивающая КС–грамматика ей эквивалентна?

Варианты ответов:

$$\begin{aligned} \text{а) } G_1: \quad & S \longrightarrow aSA|Ab|Bc \\ & A \longrightarrow aAb|ab \\ & B \longrightarrow AA|aBc \\ \text{б) } G_2: \quad & S \longrightarrow aSA|Ab|aS|b|Bc|c \\ & A \longrightarrow aAb|ab \\ & B \longrightarrow AA|aBc \\ \text{в) } G_3: \quad & S \longrightarrow aSA|aS|b|c \\ & A \longrightarrow aAb|ab \\ & B \longrightarrow aBc|ac \\ \text{г) } G_4: \quad & S \longrightarrow aSA|Ab|aS|b|Bc|c \\ & A \longrightarrow aAb|ab \\ & B \longrightarrow AA|aBc|ac|A \end{aligned}$$

д) Для заданной КС–грамматики нельзя построить эквивалентную неукорачивающую.

Правильный ответ: г.

5. Какой язык из перечисленных ниже не является контекстно–свободным:

а) $(a^*c)^n ba^n, n \geq 0$;

б) $(a^*c)^n b^{n+3} \cup a^{2n}, n > 0$;

в) $a^*c^*b^*$;

г) $a^n b^n \cup c^n, n > 0$;

д) $a^n c^n b^n a^n, n \geq 0$.

Правильный ответ: д.

Глава 2

ЯЗЫКИ И АВТОМАТЫ

2.1 Понятие автомата и типы автоматов

Автомат — это алгоритм, определяющий некоторое множество и, возможно, преобразующий это множество в другое множество. Частным случаем автомата является машина Тьюринга. Машина Тьюринга имеет управляющее устройство, которое находится в некотором состоянии из конечного множества состояний, и бесконечную ленту, предназначенную для хранения информации. Сразу отметим, что лента машины Тьюринга используется для нескольких целей:

а) перед началом работы на ней записаны исходные данные;

б) в процессе работы лента используется как рабочая память, где хранятся необходимые для работы данные;

в) после завершения работы на ленте находится результат вычислений.

Учитывая различный характер использования ленты машины Тьюринга, будем считать, что в общем случае автомат может иметь три ленты: для исходных данных, для результатов и рабочую ленту - см. рис. 2.1. Рассмотрим сначала неформальное понятие автомата в общем виде.

Автомат имеет входную ленту, управляющее устройство с конечной памятью для хранения номера состояния из некоторого конечного множества состояний, может иметь вспомогательную (или рабочую) ленту, а также может иметь выходную ленту. Автомат без выхода часто называется распознавателем, автомат с выходом — преобразователем.

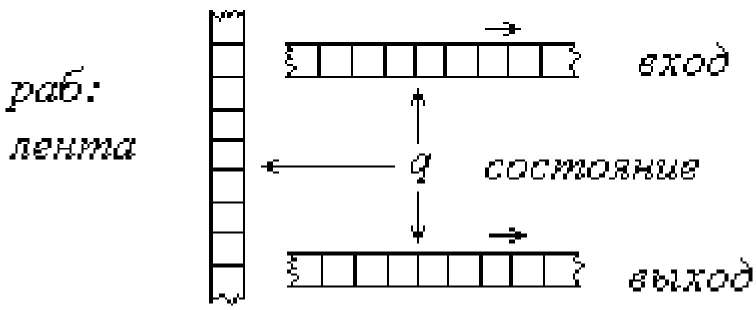


Рис. 2.1: Схематическое изображение автомата.

Входную ленту можно рассматривать как линейную последовательность ячеек, причем каждая ячейка содержит точно один символ из некоторого конечного входного алфавита. Лента бесконечна, но занято на ней в каждый момент только конечное число ячеек. Самую левую и самую правую ячейки из занятой области могут занимать специальные концевые маркеры; маркер может стоять только на правом конце ленты; маркеров может не быть совсем.

Входная головка в каждый момент времени читает (или, как иногда говорят, обозревает) одну ячейку входной ленты. За один шаг работы автомата входная головка может сдвинуться на одну ячейку вправо или остаться неподвижной. Входная головка только читает символы с входной лента, т.е. в процессе работы автомата символы на входной ленте не меняются. Более того, входная головка не может возвращаться к уже прочитанному символу. Чтение символа с входной ленты всегда означает сдвиг головки на один символ вправо.

Так же как и входная лента, рабочая и выходная ленты представляют собой последовательность ячеек, в каждой из которых может находиться только один символ некоторого алфавита. Рабочая лента — вспомогательное хранилище информации. Данные с рабочей ленты могут читаться автоматом, могут и записываться на нее. Именно сложность рабочей ленты определяет сложность автомата. Чем проще рабочая лента, тем проще автомат. Очевидно, что самый алгоритмически простой автомат не иеет рабочей ленты совсем.

Управляющее устройство — это программа, управляющая поведением автомата. Управляющее устройство представляет собой конечное множество состояний вместе с отображением, которое описывает, как меняются состояния в соответствии с текущим входным символом, читаемым входной головкой, и текущей информацией, извлеченной с рабочей ленты. Управляющее устройство определяет также, в каком направлении сдвинуть рабочую и входную головки и какую информацию записать на рабочую ленту.

Автомат работает, выполняя некоторую последовательность тактов. В начале такта читается текущий входной символ и исследуется информация на рабочей ленте. В зависимости от прочитанной информации и текущего состояния определяются действия автомата:

1) входная головка сдвигается на одну позицию вправо или остается в исходном положении;

2) на рабочую ленту может записываться некоторая информация;

3) изменяется состояние управляющего устройства;

4) на выходную ленту (если она имеется) может записываться символ.

Поведение автомата удобно описывать в терминах конфигураций автомата. Конфигурация — это как бы мгновенный снимок автомата, на котором изображены:

1) состояние управляющего устройства;

2) содержимое входной ленты с положением входной головки;

3) содержимое рабочей ленты вместе с положением рабочей головки;

4) содержимое выходной ленты, если она есть.

Управляющее устройство может быть недетерминированным и детерминированным. Если управляющее устройство недетерминированное, то для каждой конфигурации существует конечное множество возможных следующих тактов, любой из которых автомат может сделать, исходя из этой конфигурации. Управляющее устройство называется детерминированным, если для каждой конфигурации существует не более одного возможного следующего такта.

Как уже отмечалось, сложность рабочей ленты определяет сложность автомата. Обычно рассматривают следующую иерархию автоматов по уровням сложности:

— *машина Тьюринга*, имеющая бесконечную рабочую ленту, по которой головка может перемещаться в произвольном направлении, считывая содержимое ячеек ленты и записывая туда новые символы; при этом нет никаких ограничений ни на длину рабочей ленты, ни на способ доступа к ее ячейкам;

— *линейно–ограниченный автомат*, который по способу доступа к ячейкам рабочей ленты не отличается от машины Тьюринга, но имеет ограничение на длину рабочей ленты: если на вход автомата поступает цепочка x длины $|x|$, то длина рабочей ленты ограничена линейной функций от $|x|$;

— *автомат с магазинной памятью* (МП–автомат), у которого в качестве рабочей ленты используется магазин — память, работающая по принципу LIFO (Last In — First Out, или последний записанный — первый считанный);

— *конечный автомат* (КА), не имеющий рабочей ленты.

Автоматом, обладающим наибольшей алгоритмической сложностью (как эквивалент алгоритма в соответствии с тезисом Тьюринга), является машина Тьюринга. В главе 2 мы рассматривали машину Тьюринга с одной лентой, которая в начальный момент является входной, в заключительном состоянии является выходной, в процессе работы используется как рабочая. Такая машина Тьюринга эквивалентна машине Тьюринга с расщепленной лентой на три ленты в соответствии с их назначением : входная, выходная, рабочая.

Автомат без выходной ленты называется *распознавателем*, т.к. он только проверяет правильность исходных данных, т.е. распознает, принадлежит ли входная цепочка заданному множеству L или нет.

Автомат с выходной лентой является *преобразователем*, т.к. входную цепочку x при условии $x \in L$ этот автомат преобразует в новую цепочку y некоторого другого языка L_1.

Сложность автомата уменьшается с уменьшением сложности рабочей ленты: у машины Тьюринга лента неограничена в обе стороны, у линейно–ограниченного автомата длина рабочей ленты является линейной функцией длины входной цепочки, у МП–автомата рабочая лента работает по принципу магазина, ограничивая тем самым направление чтение и записи на ленту, у конечного автомата рабочая лента отсутствует. В пределах данной главы мы будем иметь дело только с распознавателями и будем называть их для краткости просто автоматами.

2.2 Формальное определение автомата

Если мы рассмотрим автоматы без рабочей ленты (конечные автоматы), то поведение такого автомата определяется только его состояниями и входными символами. Тогда для того, чтобы задать конечный автомат, необходимо определить множество его состояний K, входной алфавит X и функцию переходов как отображение $K \times X$ в множество K. В зависимости от того, зафиксировано или нет начальное состояние автомата и множество его конечных состояний, имеет автомат выходную ленту или нет, можно рассматривать различные типы автоматов.

Если введем в рассмотрение автомат с рабочей лентой, то поведение такого автомата принципиально отличается от поведения автомата без рабочей ленты. Во–первых, дополнительно необходимо определить алфавит рабочей ленты Z, во–вторых, функция переходов существенно сложнее, т.к. поведение автомата в этом случае определяется не только текущим состоянием и входным символом, но и содержимым рабочей ленты.

Определение 2.1. Неинициальный конечный автомат с выходом — это пятерка

$(K, X, Y, \delta, \gamma)$, где K, X, Y — алфавиты (называемые соответственно множеством состояний, входным и выходным алфавитом), δ — функция переходов — отображение $K \times X \to K, \gamma$ — функция выходов — отображение $K \times X \to Y$.

Функционирование автомата можно задать множеством команд вида $qx \to py$, где $q, p \in K, x \in X$ — входной символ, $y \in Y$ — выходной символ.

Пусть на некотором такте t_i блок управления находится в состоянии q и с входной ленты читается символ x. Если в множестве команд имеется команда $qx \to py$, то на том же такте t_i на выходную ленту записывается символ y, а к следующему такту t_{i+1} блок управления перейдет в состояние p. Если команды $qx \to py$ в множестве команд нет, то автомат оказывается блокированным, т.е. он никак не реагирует на символ, принятый в момент t_i, а также перестает воспринимать символы, подаваемые на вход в последующие моменты.

В соответствии с определением 2.1 в начальный момент состояние автомата может быть произвольным. Если зафиксировано некоторое начальное состояние, то такой автомат называется инициальным автоматом:

$$\begin{aligned} q(0) = \quad & q_0, \\ q(t+1) = \quad & \delta(q(t), x(t)), \\ y(t) = \quad & \gamma(q(t), x(t)), \end{aligned}$$

где $q(i) \in K$ — состояние на такте $i, x(i) \in X$ и $y(i) \in Y$ — соответственно входные и выходные символы на такте i.

Мы будем рассматривать, как правило, инициальные автоматы, поэтому основным будем считать следующее определение автомата.

Определение 2.2.

Инициальный конечный автомат с выходом — это шестерка

$$A = (K, X, Y, \delta, \gamma, q_0, F),$$

где q_0 — начальное состояние, F — множество заключительных состояний, а остальные элементы имеют тот же смысл, что и в определении 2.1.

Формальное определение автомата с рабочей лентой, кроме алфавита рабочей ленты Z, должно фиксировать вид функции переходов как частный случай отображения $K \times X \times Z$ в множество $K \times Z \times R$, где R — множество, определяющее направление движения головки по рабочей ленте.

Указанные определения соответствуют детерминированной функции переходов. Можно как для конечных автоматов, так и для автоматов более общего вида перейти к недетерминированным автоматам, рассматривая для конечных автоматов отображение $K \times X \to 2^K$ и для автоматов более обшего вида отображение $K \times X \times Z \to 2^{K \times Z \times R}$.

Таким образом, можно дать следующие два определения машины Тьюринга с расщепленной лентой, первое из которых является эквивалентом определения из главы 2, а второе определяет недетерминированную машину Тьюринга. Оба варианта определения рассмотрим для инициального автомата.

Определение 2.3. Инициальной детерминированной машиной Тьюринга называется

$$T = (K, X, Y, Z, R, \delta, \gamma, q_0, F), \text{ где}$$

K, X, Y, Z, R — алфавиты (называемые соответственно множеством состояний, входным и выходным алфавитом, алфавитом рабочей ленты и множеством направлений движения головки по рабочей ленте), δ — функция переходов — отображение $K \times X \times Z \to K \times Z \times R$, γ — функция выходов — отображение $K \times X \times Z \to Y$.

Определение 2.4. Инициальной недетерминированной машиной Тьюринга называется

$$T = (K, X, Y, Z, R, \delta, \gamma, q_0, F), \text{ где}$$

K, X, Y, Z, R имеют тот же смысл, что в определении 2.3, а δ — функция переходов — отображение $K \times X \times Z \to 2^{K \times Z \times R}$, γ — функция выходов — отображение $K \times X \times Z \to 2^Y$.

Задача грамматического разбора заключается в нахождении вывода заданной цепочки в заданной грамматике и в построении дерева вывода этой цепочки. Для решения этой задачи будем использовать автоматы.

Языки могут быть заданы двумя способами: грамматиками и автоматами. Грамматики являются порождающим способом определения языка, т.к. с помощью грамматик формализуется способ порождения (или вывода) всех цепочек языка. Автоматы являются распознающим способом формализации языка, так с помощью автомата можно определить множество цепочек, которое распознает этот автомат при переходе из начального состояния в заключительное. Поэтому, если автоматы используются для определения языков, то необходимо рассматривать инициальные автоматы.

В данной главе мы покажем, что различным по сложности автоматам соответствуют разные типы языков. Простейшим типом автоматов являются конечные автоматы. Докажем, что этим автоматам соответствуют линейные грамматики — леволинейные и праволинейные. Как уже отмечалось, для определения синтаксиса языков программирования обычно используются КС–грамматики. Докажем, что КС–грамматикам соответствуют МП–автоматы.

2.3 Конечные автоматы

Конечный автомат имеет входную ленту, с которой за один такт может быть прочитан один входной символ. Возврат по входной ленте не допускается, как, впрочем, он не допускается обычно для любого типа автоматов с разделенной по функциональному назначению лентой. В отличие от предшествующего параграфа, где было введено определение автомата в общем виде, сейчас мы перейдем к определению конечного инициального автомата–распознавателя.

Определение 2.5. Конечным автоматом называется шестерка вида

$$A = (K, \Sigma, \delta, p_0, F), \text{ где}$$

K — конечное множество состояний,

Σ — алфавит,

δ — функция переходов, в общем случае — недетерминированное отображение $\delta : K \times \Sigma \to 2^K$,

p_0 — начальное состояние, $p_0 \in K$,

F — множество заключительных состояний, $F \subseteq K$.

Частным случаем конечных автоматов являются детерминированные конечные автоматы с функцией переходов $\delta : K \times \Sigma \to K$.

Любой автомат, в том числе и конечный, можно определить как формальную систему через состояния, символы, которые пишутся или читаются с ленты или нескольких лент, и набора команд. В частности, конечный автомат можно представить командами, графом, таблицей переходов и матрицей преходов.

Пример 2.1. Автомат, распознающий язык a^*bc^*, может быть представлен следующим списком команд:

$$p_0, a \to p_0,$$
$$p_0, b \to p_1,$$
$$p_1, c \to p_1,$$

где p_0 — начальное состояние, p_1 — заключительное состояние.

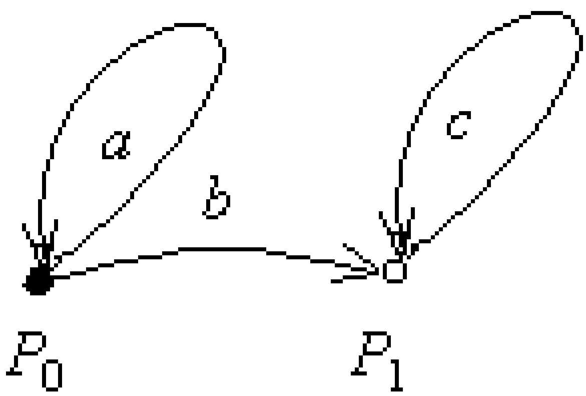
Этот же автомат можно задать матрицей переходов:

	p_0	p_1
p_0	a	b
p_1		c

Эквивалентное задание автомата таблицей переходов имеет вид:

	a	b	c
p_0	p_0	p_1	
p_1			p_1

Этот же автомат может быть представлен графом:



2.4 Регулярные множества

Определение 2.6. Язык, распознаваемый конечным автоматом

$$A = (K, \Sigma, \delta, p_0, F)$$

— это множество цепочек, читаемых автоматом A при переходе из начального состояния в одно из заключительных состояний:

$$L(A) = \{a_1 a_2 ... a_n | p_0 a_1 \to p_1, ..., p_{n-1} a_n \to p_n; p_n \in F\}$$

Определение 2.7. Автоматы называются эквивалентными, если совпадают распознаваемые ими языки.

Определение 2.8. Множество называется регулярным, если существует конечный детерминированный автомат, распознающий это множество.

Теорема 2.1. Для любого конечного автомата распознаваемое множество регулярно.

Доказательство. Формулировка теоремы означает, что для любого недетерминированного конечного автомата можно построить эквивалентный детерминированный автомат. Выполним такое построение. Пусть $A = (K, \Sigma, \delta, p_0, F)$ — произвольный недетерминированный конечный автомат. Построим новый автомат

$$A_1 = (K_1, \Sigma, \delta_1, [p_0], F_1) \ .$$

Множество состояний автомата A_1 состоит из всевозможных подмножеств состояний исходного автомата, причем, если хотя бы одно состояние исходного автомата A, принадлежащее составному состоянию автомата A_1, является заключительным, то новое составное состояние является заключительнвм для автомата A_1:

$$\begin{aligned} K_1 = & \quad \{[p_{i_1}, ..., p_{i_j}] | p_{i_1}, ..., p_{i_j} \in K\}, \\ F_1 = & \quad \{[p_{i_1}, ..., p_{i_k}] | p_{i_j} \in F\}. \end{aligned}$$

Функция переходов δ_1 формируется следующим образом : для каждого состояния $[p_{i_1}, ..., p_{i_k}]$ в левой части команды переход по некоторому символу a в новое состояние в правой части этой команды формируется как объединение всех состояний p_{j_m}, в которые возможен переход в A из всех $p_{i_l} (1 \leq l \leq k)$:

$$\begin{aligned} \delta_1 \quad & = \{[p_{i_1}, ..., p_{i_t}]a \to [p_{j_1}, .., p_{j_r}] | \\ & \forall p_{i_l} \exists \ p_{j_t} (p_{i_l} a \to p_{j_t} \in \delta), \ \forall p_{j_t} \exists \ p_{i_l} (p_{i_l} a \to p_{j_t} \in \delta)\} \end{aligned}$$

В соответствии с определением языка, распознаваемого автоматом, для любой цепочки $x \in L(A)$ существует последовательность команд автомата A, переводящая его из начального состояния в заключительное при чтении цепочки x. Для каждой такой команды в δ найдется одна и только одна команда δ_1, читающая тот же символ, следовательно A_1 читает все цепочки, которые читает исходный автомат A. Обратно: для любой цепочки, читаемой автоматом A_1 по построению δ_1 существует одна или несколько последовательностей команд автомата A, выполняющая те же действия. Таким образом, $L(A) = L(A_1)$. $\square$

Для реализации рассмотренного алгоритма удобно использовать таблицу переходов автомата. Для простоты новые состояния лучше обозначать не множествами, а символами с индексами.

Очевидно, что алгоритм, рассмотренный в доказательстве теоремы 2.1, генерирует полное множество состояний, так что $|K_1| = 2^{|K|}$. В множество K_1 входят все возможные состояния, в том числе и не достижимые из начального состояния. Чтобы такие состояния не рассматривать, лучше последовательно строить переходы для всех вновь появляющихся состояний, начиная от состояния $[P_0]$.

Пример 2.2. Пусть исходная таблица переходов конечного автомата имеет вид:

	a	b	c	
P_0	P_0		P_0, P_1	нач.
P_1	P_1, P_3	P_0		
P_2			P_1	закл.
P_3		P_2	P_0	закл.

Выполняя операции объединения состояний, получим

	a	b	c	
P_0	P_0		P_{01}	нач.
P_{01}	P_{013}	P_0	P_{01}	
P_{013}	P_{013}	P_{02}	P_{01}	закл.
P_{02}	P_0		P_{01}	закл.

2.5 Минимизация конечных автоматов

Определение 2.9. Два состояния p_1 и p_2 конечных автоматов соответственно A_1 и A_2 называются n–эквивалентными, если, начиная действие из этих состояний, автомат распознает совпадающие множества цепочек длины не более n.

Сразу следует отметить, что в этом определении для общности рассматриваются два автомата. Однако, ничто не препятствует рассматривать и два состояния одного автомата. В этом случае появляется возможность сравнивать состояния одного автомата и, возможно, заменять их на одно состояние.

Пример 2.3. Рассмотрим два автомата, представленных на рис. 2.2.

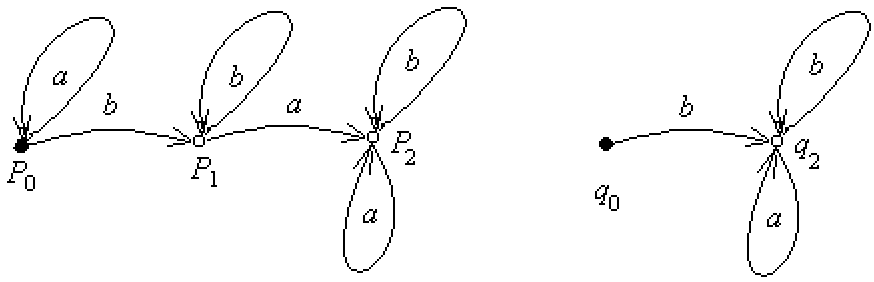


Рис. 2.2: Автоматы с 2–эквивалентными состояниями.

Пусть $n = 2$, тогда, например, из состояний p_0, q_0, p_1, q_2 распознаются следующие множества цепочек длины не более n:

$$\begin{array}{ll} p_0 \Rightarrow b, ab, bb, ba, & q_0 \Rightarrow b, ba, bb, \\ p_1 \Rightarrow b, a, bb, ba, ab, aa, & q_2 \Rightarrow b, a, bb, ba, ab, aa. \end{array}$$

Состояния p_1 и q_2 являются 2–эквивалентными, а, например, состояния p_0 и q_0 не являются 2–эквивалентными, но являются 1–эквивалентными.

Определение 2.10. Два состояния p_1 и p_2 конечных автоматов соответственно A_1 и A_2 называются эквивалентными, если они n–эквивалентны для любого n.

Определение 2.11. Конечный автомат называется минимальным, если никакие его два состояния не эквивалентны друг другу.

Теорема 2.2. Для любого конечного автомата $A = (K, \Sigma, \delta, p_0, F)$ можно построить эквивалентный минимальный автомат, выполняя не более $|K|$ разбиений матрицы переходов.

Доказательство. Доказательство основано на выделении групп n–эквивалентных состояний. Пусть имеется матрица переходов автомата A. Сразу можно заметить, что все заключительное и все незаключительное состояния не эквивалентны друг другу, т.к. в заключительном состоянии может закончиться процесс распознавания, а в незаключительном он обязан продолжиться. Таким образом, в заключительном состоянии распознается пустая цепочка ε, а в незаключительном состоянии

— пустое множество цепочек. Это означает 0–неэквивалентность заключительных и незаключительных состояний. Тогда уже на первом шаге алгоритма определения эквивалентных состояний мы можем разбить состояния на две группы: заключительные и незаключительные состояния как не эквивалентные для $n = 0$.

Пусть теперь построены группы n–эквивалентных состояний $H_1, H_2, ..., H_t$. В соответствии с определением, из каждого состояния $p_k \in H_i$ распознается одно и то же множество цепочек M_i длины не более n. Перейдем к анализу $(n+1)$–эквивалентности.

Если все группы эквивалентности содержат точно по одному состоянию, то любая пара сотояний не эквивалентна друг другу, исходный автомат является минимальным. Пусть хотя бы одна группа n–эквивалентных состояний H_i содержит более одного состояния. Рассмотрим все переходы из $p_k \in H_i$ и из $p_m \in H_i$ в соответствии с матрицей переходов как внутри группы эквивалентности H_i, так и в некоторую другую группу H_j. Для того, чтобы состояния p_k и p_m оставались $(n+1)$–эквивалентными, необходимо и достаточно, чтобы переход в каждую группу n–эквивалентности H_j осуществлялся по одним и тем же символам $\{a_{j1}, a_{j2}, ...a_{jr}\}$: тогда для каждого из этих состояний допускается одно и то же множество цепочек $\{a_{j1}, a_{j2}, ...a_{jr}\} \cdot M_j$. Другими словами, в каждой подматрице, соответствующей группе эквивалентности, каждая строка должна содержать одни и те же символы. Если строки хотя бы одной подматрицы содержат разные символы, то соответствующие состояния не являются $(n+1)$–эквивалентными. Разбиение групп n–эквивалентных состояний на подгруппы $(n+1)$–эквивалентных состояний закончится не более чем за $|K|$ шагов, т.к. на каждом шаге отделяется по меньшей мере одно неэквивалентное состояние от какой–либо группы. $\square$

В соответствии с доказательством теоремы можно предложить следующий *алгоритм построения минимального автомата*, эквивалентного исходному.

1. Строится матрица переходов конечного автомата.

2. В матрице группируются отдельно заключительные и незаключительные состояния. Между ними проводится граница как по строкам, так и по столбцам.

3. Рассматриваются поочередно все подматрицы полученной матрицы. В каждой строке такой подматрицы должны быть одинаковые символы, что означает переход либо между подгруппами, либо внутри группы по одному и тому же символу. Если строки содержат разные символы, их нужно сгруппировать так, чтобы в каждой подгруппе содержались одни и те же символы и выполнить разбиение по границам между группами.

4. Повторяется п.4 до тех пор, пока возможно разбиение.

5. Когда разбиение закончено, каждой группе эквивалентности сопоставляется одно состояние.

Пример 2.4. Дана следующая матрица переходов:

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	
p_1		a				a d		нач
p_2	b				b	a	b	
p_3				b		a b		
p_4		a	a			a	d	
p_5			a			d	a	
p_6		c	c					закл
p_7			c					закл

Применим алгоритм минимизации и получим разбиение этой матрицы

	p_1	p_4	p_5	p_2	p_3	p_6	p_7	
p_1				a		a d		нач
p_4				a	a	a	d	
p_5					a	d	a	
p_2	b		b			a	b	
p_3		b				a b		
p_6				c	c			закл
p_7					c			закл

В каждой подматрице получили одинаковые строки, поэтому каждой группе эквивалентных состояний поставим в соответствие одно состояние нового автомата:

	p_1	p_2	p_3	
p_1		a	a d	нач
p_2	b		a b	
p_3		c		закл

2.6 Операции над регулярными языками

Так как произвольному конечному автомату однозначно соответствует детерминированный конечный автомат, операции над конечными автоматами эквивалентны операциям над регулярными множествами.

Известно, что для произвольного конечного автомата можно построить эквивалентный автомат без циклов в начальном и (или) конечных состояниях. Для построения алгоритмов таких преобразований рассмотрим следующие теоремы.

Теорема 2.3. Для произвольного конечного автомата существует эквивалентный автомат без циклов в начальном состоянии.

Доказательство. Пусть $A = (K, \Sigma, \delta, p_0, F)$ — произвольный конечный автомат. Построим новый конечный автомат

$$A_1 = (K \cup \{q_0 | q_0 \notin K\}, \Sigma, \delta \cup \{q_0 a \to p_i | p_0 a \to p_i \in \delta\}, q_0, F \cup \{q_0 | p_o \in F\}).$$

Рассмотрим сначала цепочку ε. Цепочка $\varepsilon \in L(A)$ тогда и только тогда, когда $p_0 \in F$, но в этом случае по построению автомата A_1 его начальное состояние q_0 является заключительным и, следовательно, $\varepsilon \in L(A_1)$.

Перейдем теперь а анализу цепочек x общего вида: $x \neq \varepsilon$. По построению автомат A не имеет циклов в начальном состоянии q_0. Любая цепочка $x = a_1, a_2, ..., a_k \neq \varepsilon$ принадлежит $L(A)$ тогда и только тогда, когда существует последовательность команд автомата A

$$p_0 a_1 \to p_1\ ,\ p_1 a_2 \to p_2\ , \ldots ,\ p_{k-1} a_k \to p_k\ ,\ p_k \in F$$

и соответствующая ей последовательность команд автомата A_1

$$q_0 a_1 \to p_1\ ,\ p_1 a_2 \to p_2\ , \ldots ,\ p_{k-1} a_k \to p_k\ .$$

Следовательно, $L(A) = L(A_1)$.□

Теорема 2.4. Для произвольного конечного автомата существует эквивалентный автомат без циклов в заключительном состоянии.

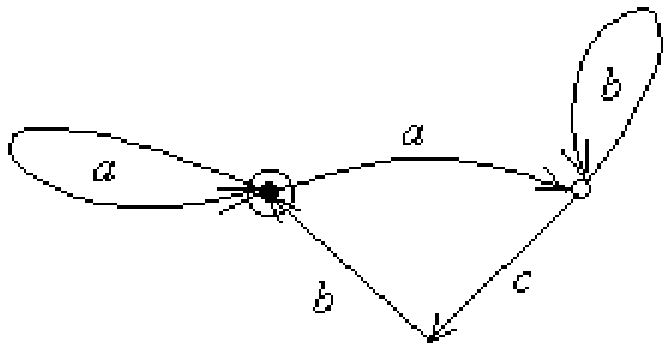


Рис. 2.3: Конечный автомат с циклами в начальном и заключительном состояниях.

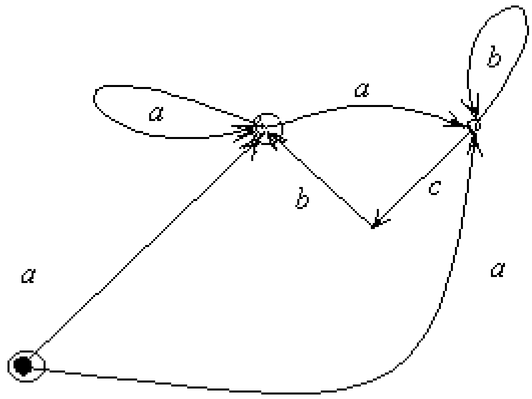


Рис. 2.4: Конечный автомат без циклов в начальном состоянии.

Доказательство. По теореме 2.3 можем считать, что исходный автомат $A = (K, \Sigma, \delta, p_0, F)$ не имеет циклов в начальном состоянии. Сопоставим заданному произвольному конечному автомату A новый автомат

$$A_1 = (K \cup \{f | f \notin K\}, \Sigma, \delta \cup \{p_j a \to f | p_j a \to p_i \in \delta \& p_i \in F\}, p_0, \{f\} \cup \{p_0 | p_0 \in F\}).$$

По построению A не имеет циклов в заключительном состоянии f. Если заключительным состоянием является также и состояние p_0, то циклы в этом состоянии отсутствуют по условию предварительного преобразования исходного автомата по теореме 2.4. Эквивалентность $L(A) = L(A_1)$ очевидна. □

Пример 2.5. Построим автомат без циклов в начальном и заключительном состояниях эквивалентный автомату, представленному на рис 2.3.

Предварительно удалим циклы, проходящие через начальное состояние. Получим автомат, представленный на рис. 2.4.

Затем удалим циклы, проходящие через заключительные состояния. Таких состояний три, однако, начальное состояние, одновременно являющееся и заключительным, от циклов свободно. В результате получим автомат, представленный на рис 2.5.

Рассмотрим теперь операции над регулярными множествами или, что то же самое, над языками, допускаемыми конечными автоматами.

Теорема 2.5. Множество регулярных языков замкнуто относительно операций

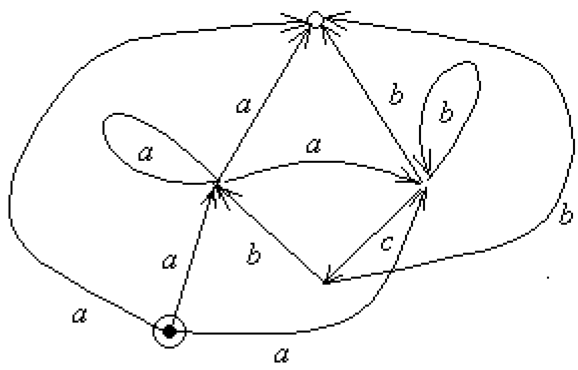


Рис. 2.5: Конечный автомат без циклов в заключительном состоянии.

итерации, усеченной итерации, произведения, объединения, пересечения, дополнения, разности.

Доказательство. Для доказательства необходимо выполнить операции над соответствующими конечными автоматами и показать, что в результате таких преобразований построенный автомат допускает требуемый язык.

Поскольку можно удалить циклы из начальных и заключительных состояний любого автомата, всегда будем предполагать, что такие преобразования сделаны, если это необходимо. Тогда для выполнения указанных в теореме операций необходимо выполнить соответствующие преобразования над заданными автоматами.

Операция итерации реализуется удалением циклов из начальных и конечных состояний и объединением этих полученных состояний. Действительно, объединение начального и заключительного состояний означает, что построенный автомат допускает цепочку ε. Однократный переход из начального в заключительное состояние исходного автомата соответствует допуску цепочек языка L. Поскольку эти состояния объединены, построенный автомат допускает цепочки языков LL, LLL и т.д., т.е. он распознает язык $\{\varepsilon\} \cup L \cup L^2 \cup \cdots = L^*$.

Операция произведения над $L(A_1)$ и $L(A_2)$ выполняется с помощью двух преобразований:

a) удалим циклы из заключительного состояния A_1 или из начального состояния A_2;

b) каждому заключительному состоянию p_i автомата A_1 поставим в соответствие свой экземпляр автомата A_2 и объединяем каждое заключительное состояние автомата A_1 с начальным состоянием соответствующего экземпляра автомата A_2.

Объединение $L(A_1)$ и $L(A_2)$ строится с помощью удаления циклов в начальных состояниях A_1 и A_2 и объединения полученных начальных состояний.

Усеченная итерация может быть построена как произведение вида

$$L(A_1)^+ = L(A_1)^* L(A_1)$$

или вида

$$L(A_1)^+ = L(A_1) L(A_1)^*.$$

Соответствующие операции над автоматами мы уже рассмотрели.

Рассмотрим дополнение $L(A_1)$ до Σ^*. Допустим, автомат A_1 является детерминированным, т.к. по теореме 2.1 любой автомат можно преобразовать к эквивалентному детерминированному. Известно, что автомат можно задать графом переходов, тогда любая цепочка $x = a_1a_2..a_n$ распознается детерминированным автоматом A_1 по единственному маршруту:

$$\begin{array}{ll} P_0a_1 \to & P_{i_1} \\ P_{i_1}a_2 \to & P_{i_2} \\ \dots & \\ P_{i_{n-1}}a_n \to & P_z, \text{ где } P_z \in F\ . \end{array}$$

Автомат A_1 не распознает те и только те цепочки, которые

a) либо представляют собой начальную часть цепочки $a_1a_2...a_j$, при чтении которой автомат переходит в состояние, не являющееся заключительным;

b) либо имеют вид $y = a_1a_2...a_lbc_1c_2...c_r, l \le n$, где начало $a_1a_2...a_l$ совпадает с началом цепочки $x \in L(A_1)$, но за символом a_l стоит такой символ b, что автомат A_1 его прочитать не может.

Следовательно, для того, чтобы построить автомат, распознающий дополнение языка $L(A)$, надо для детерминированного конечного автомата A выполнить следующие действия:

a) все заключительные состояния сделать незаключительными, а все незаключительные — заключительными;

b) ввести дополнительное состояние q, сделать его заключительным и из каждого состояния p_i провести в новое состояние q такие дуги $\overrightarrow{p_iq}$, каждая из которых соответствует символам алфавита, не читаемым в состоянии p_i;

c) в построенном дополнительном состоянии q построить петли для всех символов алфавита, чтобы обеспечить чтение произвольного окончания цепочки $c_1c_2...c_r$.

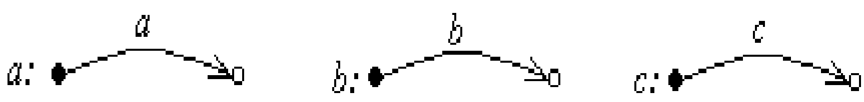
Оставшиеся операции разности и пересечения регулярных языков можно определить через уже рассмотренные операции с помощью следующих тождеств:

$$\begin{array}{ll} L(A_1)\backslash L(A_2) = & L(A_1) \cap \overline{L(A_2)} \\ L(A_1) \cap L(A_2) = & \overline{\overline{L(A_1)} \cup \overline{L(A_2)}} \end{array}$$

□

Рассмотренная теорема предлагает алгоритмы построения конечных автоматов, позволяя последовательно синтезировать автоматы на базе уже построенных.

Пример 2.6. Построить конечный автомат, распознающий язык $a^*b \cup b^+c$. Для решения этой задачи последовательно построим автоматы, начиная от простейших и заканчивая автоматом, распознающим заданный язык в целом. Сначала построим автоматы, которые распознают элементарные языки, состоящие из единственной цепочки.



Затем выполним операцию итерации и операцию усеченной итерации соответственно над языками $\{a\}$, $\{b\}$. Получим автоматы, представленные на рис. 2.6.

Построим a^*b как произведение a^* на b. Удалим циклы из начального состояния полученного автомата, получим автомат, представленный на рис. 2.7.

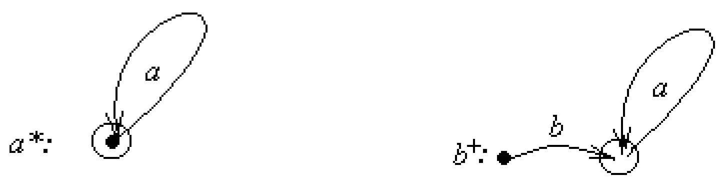


Рис. 2.6: Конечные автоматы, распознающие a^* и b^+.

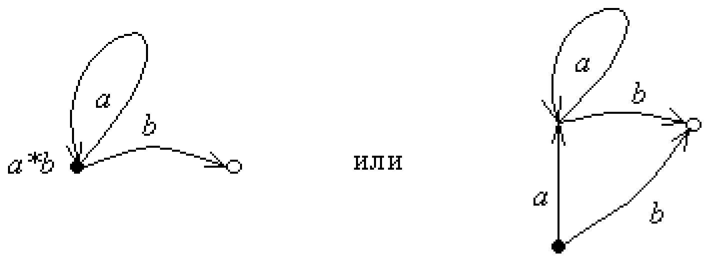


Рис. 2.7: Конечный автомат, распознающий a^*b.

Построим произведение b^+ на c, получим автомат, представленный на рис. 2.8.

Тереръ осталось построить объединение языков a^*b и b^+c.. Начальные состояния соответствующих автоматов циклов не содержат, поэтому просто объединим эти состояния. В результате получим автомат, распознающий язык $a^*b \cup b^+c$ (см. рис. 2.9).

2.7 Автоматные грамматики и конечные автоматы

Мы рассмотрели два вида автоматных грамматик: леволинейные и праволинейные с правилами вида $A \to Ba, A \to a$ или $A \to aB, A \to a$ соответственно. Покажем, что языки, порождаемые линейными грамматиками, совпадают с языками, распознаваемыми конечными автоматами.

Теорема 2.6. Для каждой праволинейной грамматики существует эквивалентный конечный автомат.

Доказательство. Каждому нетерминальному символу $A_i \in V_N$ произвольной праволинейной грамматики $G = (V_T, V_N, P, A_0)$ поставим в соответствие состояние A_i конечного автомата A. Добавим еще одно состояние F и сделаем его единственным конечным состоянием. Состояние, соответствующее аксиоме, сделаем начальным.

Каждому правилу $A_i \to aA_j$ поставим в соответствие команду $A_i, a \to A_j$ автомата A, а каждому терминальному правилу $A_i \to a$ — команду $A_i, a \to F$. Тогда каждому выводу в грамматике

$$A_0 \Rightarrow a_1A_{i_1} \Rightarrow a_1a_2A_{i_2} \Rightarrow \ldots \Rightarrow a_1a_2...a_{k-1}A_{i_{k-1}} \Rightarrow a_1a_2...a_{k-1}a_k$$

взаимно–однозначно соответствует последовательность команд построенного авто-

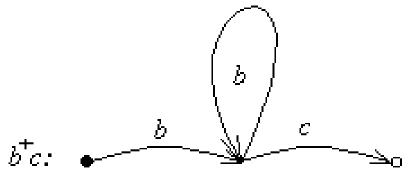


Рис. 2.8: Конечный автомат, распознающий b^+c.

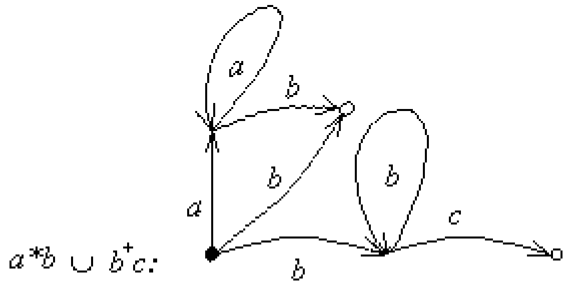


Рис. 2.9: Конечный автомат для примера 2.6.

мата

$$\begin{array}{l} A_0, a_1 \to A_{i_1}, \\ A_{i_1}, a_2 \to A_{i_2}, \\ \dots \\ A_{i_{k-2}}, a_{k-1} \to A_{i_{k-1}}, \\ A_{i_{k-1}}, a_k \to F. \end{array}$$

Следовательно, $L(G) = L(A)$. $\square$

Пример 2.7. Для заданной грамматики

$$G: \begin{array}{l} S \to aS|bB \\ A \to aA|bS \\ B \to bB|c|cA \end{array}$$

эквивалентный конечный автомат представлен на рис. 2.10.

Теорема 2.7. Для произвольного конечного автомата существует эквивалентная праволинейная грамматика.

Доказательство. Каждому состоянию p_i произвольного конечного автомата $A = (K, \Sigma, \delta, p_0, F)$ поставим в соответствие нетерминальный символ P_i грамматики, причем начальному состоянию p_0 поставим в соответствие аксиому. Тогда для каждой команды $p_i, c \to p_j$ в множество правил грамматики включим правило $P_i \to cP_j$, причем если P_j — заключительное состояние, то добавим правило $P_i \to c$. Эквивалентность исходного автомата и построенной грамматики очевидна. $\square$

Теорема 2.8. Для каждой леволинейной грамматики существует эквивалентный конечный автомат.

Доказательство. Каждому нетерминальному символу A_i произвольной леволинейной грамматики $G = (V_T, V_N, P, A_0)$ поставим в соответствие состояние p_i конечного автомата A, причем состояние p_0, соответствующее аксиоме A_0, сделаем заключительным. Добавим еще одно состояние N и сделаем его начальным состоянием.

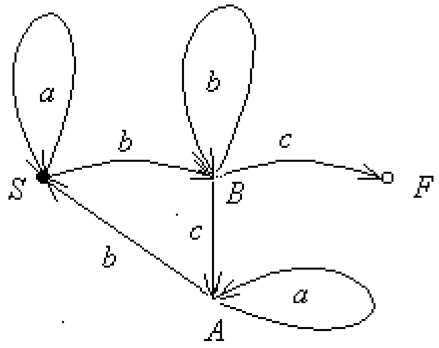


Рис. 2.10: Конечный автомат для грамматики примера 2.7.

Каждому правилу $A_i \to A_j a$ поставим в соответствие команду $p_j, a \to p_i$, а каждому терминальному правилу $A_i \to a$ — команду $N, a \to A_i$. Тогда каждому выводу в грамматике

$$A_0 \Rightarrow A_{i_1} a_k \Rightarrow A_{i_2} a_{k-1} a_k \Rightarrow \ldots \Rightarrow A_{i_{k-1}} a_2 ... a_k \Rightarrow \ldots \Rightarrow a_1 a_2 ... a_k$$

взаимо–однозначно соответствует последовательность команд построенного автомата

$$N, a_1 \to p_{i_{k-1}}, \ldots, p_{i_2}, a_{k-1} \to p_{i_1}, p_{i_1} a_k \to p_0.$$

Следовательно, $L(G) = L(A)$. □

Теорема 2.9. Для произвольного конечного автомата существует эквивалентная леволинейная грамматика.

Доказательство. Каждому состоянию p_i произвольного конечного автомата $A = (K, \Sigma, \delta, p_0, F)$ поставим в соответствие нетерминальный символ A_i грамматики. Добавим еще один нетерминал S и сделаем его аксиомой. Для каждой команды $p_i, a \to p_j$ автомата A в множество правил грамматики включим правило $A_j \to A_i a$, причем если p_j — заключительное состояние, то дополнительно сформируем правило $S \to A_i a$, а если A_i — начальное состояние, то дополнительное правило $A_j \to a$. Тогда последовательности команд

$$\begin{array}{l} p_0, a_1 \to p_{i_1}, \\ p_{i_1}, a_2 \to p_{i_2}, \\ \ldots \\ p_{i_{k-1}}, a_k \to F \end{array}$$

взаимнооднозначно соответствует вывод

$$S \Rightarrow A_{i_{k-1}} a_k \Rightarrow ... \Rightarrow A_{i_2} a_3 ... a_k \Rightarrow A_{i_1} a_2 a_3 ... a_k \Rightarrow a_1 a_2 ... a_k.$$

Языки $L(A)$ и $L(G)$ совпадают. □

Ранее (см. теорему 2.1) мы показали регулярность множеств, распознаваемых конечными автоматами. Тогда эквивалентность языков, порождаемых линейными грамматиками, и языков, распознаваемых конечными автоматами, можно сформулировать в терминах регулярных множеств.

Следствие 2.1. Языки, порождаемые линейными грамматиками, регулярны.

Следствие 2.2. Классы леволинейных и праволинейных грамматик эквивалентны.

Пример 2.8. Для заданной леволинейной грамматики построить эквивалентную праволинейную грамматику.

$$\begin{aligned} G_1: \quad & S \to Aa \\ & A \to Bc|d \\ & B \to Ba|b \end{aligned}$$

Преобразование леволинейной грамматики в праволинейную можно выполнить с помощью построения конечного автомата в качестве промежуточного шага такого преобразования. Сначала строим конечный автомат для грамматики G_1 (см. рис. 2.11.).

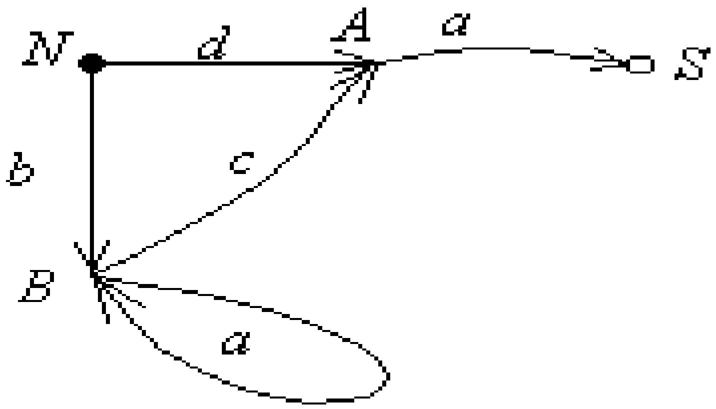


Рис. 2.11: Конечный автомат для леволинейной грамматики примера 2.8.

Этому автомату соответствует праволинейная грамматика:

$$\begin{aligned} G_2: \quad & N \to dA|bB \\ & A \to a \\ & B \to aB|cA \end{aligned}$$

Цепочка, например, $baaca$ выводится как в грамматике G_1:

$$S \Rightarrow Aa \Rightarrow Bca \Rightarrow Baca \Rightarrow Baaca \Rightarrow baaca,$$

так и в грамматике G_2:

$$N \Rightarrow bB \Rightarrow baB \Rightarrow baaB \Rightarrow baacA \Rightarrow baaca.$$

Ранее мы показали, что класс КС–языков не замкнут относительно операций пересечения. Сейчас мы докажем важный факт, позволяющий, когда это необходимо, путем пересечения с подходящим регулярным множеством отбрасывать те цепочки данного КС–языка, которые не имеют интересующей нас формы, и причем так, что остающиеся цепочки также образуют КС–язык. Это свойство подтверждает фундаментальную роль регулярных множеств в теории КС–языков.

Теорема 2.10. Пересечение КС–языка и регулярного множества является КС–языком.

Доказательство. Пусть $L(G)$ — КС–язык, порождаемый заданной грамматикой $G = (V_T, V_N, P, S)$. В силу существования эквивалентого преобразования к неукорачивающей форме можно считать G неукорачивающей грамматикой. Пусть M — регулярное множество, распознаваемое конечным автоматом $A = (K, \Sigma, \delta, p_O, F)$, причем

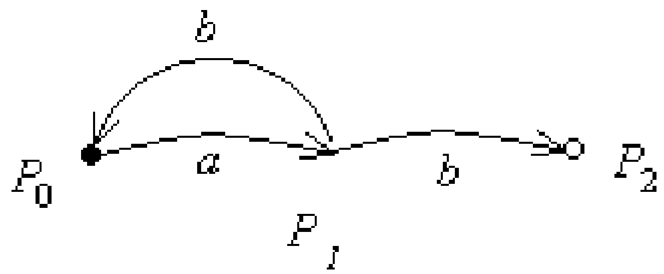


Рис. 2.12: Конечный автомат, распознающий язык $(ab)^+$.

$\varepsilon \notin M$. По теореме 2.4 можно считать, что A имеет единственное заключительное состояние и $F = \{p_z\}$.

Будем считать, $V_T = \Sigma$, т.к. любое из них всегда можно дополнить до $V_T \cup \Sigma$. Построим новую КС–грамматику

$$G^1 = (V_T, V_N^1, P^1, S^1) \ ,$$

где V_N^1 состоит из трехэлементных символов $V_N^1 = K \times V_N \times K$, и $S^1 = (p_O, S, p_z)$.

Построим множество правил грамматики G^1 так, чтобы они одновременно отслеживали правила вывода в G и элементарные такты работы автомата A в процессе распознавания входной цепочки:

1) если $B \to a_1a_2...a_k \in P, B \in V_N, a_i \in V_T \cup V_N$, то в множество P^1 включим всевозможные правила вида

$$(p_iBp_j) \to (p_ia_1p_{i_1})(p_{i_1}a_2p_{i_2})...(p_{i_{k-2}}a_{k-1}p_{i_{k-1}})(p_{i_{k-1}}a_kp_j)$$

для всех $p_i, p_j, p_{i_t} \in K$;

2) в множество P^1 включим правило $(p, a, q) \to a$, если $a \in V_T$ и $\delta(p, a) = q$, т.е. имеется команда автомата $pa \to q$.

Для каждого вывода $S \overset{*}{\Rightarrow} a_1a_2...a_t$ в грамматике G очевидно существование соответствующего вывода в G^1

$$(p_0Sp_z) \overset{*}{\Rightarrow} (p_0a_1p_{i_1})(p_{i_1}a_2p_{i_2})...(p_{i_{t-1}}a_tp_z)$$

и обратно, каждому такому выводу в G^1 соответствует вывод $S \overset{*}{\Rightarrow} a_1a_2...a_t$ в G. Далее, по построению G^1 правило $(paq) \to a$ принадлежит P^1 тогда и только тогда, когда $a \in V_T$ и $p, a \to q \in \delta$, следовательно, $p_0, p_{i_1}, p_{i_2}, ..., p_z$ должны быть теми состояниями автомата A, через которые проходит процесс распознавания цепочки $a_1a_2...a_t$. Таким образом, $a_1a_2...a_t \in L(A) \cap L(G)$ тогда и только тогда, когда $(p_0Sp_z) \overset{*}{\Rightarrow} a_1a_2...a_t$ в построенной КС–грамматике G^1, т.е. $L \cap M$ — КС–язык. $\Box$

Пример 2.9. $M = (ab)^+$, $L = a^nb^n$, $n > 0$. Ясно, что $M \cap L = ab$. Покажем этот факт средствами, используемыми при доказательстве теоремы 2.10. Действительно, язык L порождает КС–грамматика

$$G : S \to aSb|ab$$

Автомат, распознающий M, представлен на рис. 2.12.

Строим новую грамматику по правилам, применяемым при доказательстве теоремы:

$$\begin{aligned} G_1: \quad & (p_iSp_j) \to (p_iap_t)(p_tSp_k)(p_kbp_j) \\ & (p_iSp_j) \to (p_iap_t)(p_tbp_j) \\ & (p_0ap_1) \to a \\ & (p_1bp_0) \to b \\ & (p_1bp_2) \to b, \end{aligned}$$

где (p_0Sp_2) — аксиома. Так как для аксиомы имеется только два правила грамматики, то можно рассмотреть два варианта вывода терминальных цепочек языка, порождаемого грамматикой G_1. Рассмотрим самый короткий вывод из аксиомы с использованием аналога правила $S \to ab$:

$$(p_0Sp_2) \Rightarrow (p_0ap_k)(p_kbp_2) = (p_0ap_1)(p_1bp_2) \Rightarrow a(p_1bp_2) \Rightarrow ab.$$

Вывод из аксиомы с использованием аналога правила $S \to aSb$ имеет вид:

$$(p_0Sp_2) \Rightarrow (p_0ap_1)(p_1Sp_1)(p_1bp_2) \Rightarrow a(p_1ap_t)(p_tSp_k)(p_kbp_1)b \Rightarrow ...$$

Этот вывод никогда не приведет к терминальной цепочке, т.к. для любых t и k для нетерминалов (p_1ap_t) и (p_kbp_1) нет правил. Следовательно, цепочка ab —единственная цепочка, принадлежащая языку, и $L(G_1) = ab$.

2.8 Автоматы с магазинной памятью и КС–языки

В отличие от конечного автомата МП–автомат имеет рабочую ленту — магазин.

Определение 2.12. МП–автомат — это семерка вида

$$M = (K, \Sigma, \Gamma, \delta, p_0, F, B_0), \text{ где}$$

K — конечное множество состояний,
Σ — алфавит,
Γ — алфавит магазина,
δ — функция переходов,
p_0 — начальное состояние,
F — множество заключительных состояний,
B_0 — символ из Γ для обозначения маркера дна магазина.

В общем случае это определение соответствует недетерминированному автомату. В отличие от конечного автомата для произвольного недетерминированного МП–автомата нельзя построить эквивалентный детерминированный МП–автомат. Примем этот факт без доказательства.

Основное использование распознавательных средств задания языков — это построение алгоритмов грамматического разбора, тогда необходимо для произвольной КС–грамматики уметь строить эквивалентный МП–автомат. МП–автоматы представляют интерес как средство разбора в КС–грамматиках произвольного вида. Этот факт сформулирован в следующей теореме.

Теорема 2.11. Языки, порождаемые КС–грамматиками, совпадают с языками, распознаваемыми МП–автоматами.

Доказательство. Существуют две стратегии разбора: восходящая и нисходящая. При восходящей стратегии разбора необходимо найти основу и редуцировать ее в соответствии с правилами грамматики к какому–нибудь нетерминалу. Это можно сделать, если реализовать следующий алгоритм функционирования МП–автомата:

a) любой входной символ записывается в магазин;

b) если в верхушке магазина сформирована основа, совпадающая с правой частью правила, то она заменяется на нетерминал в левой части этого правила;

c) разбор заканчивается, если в магазине аксиома, а входная цепочка просмотрена полностью.

В соответствии с указанным алгоритмом для КС–грамматики $G = (V_t, V_n, P, S)$ построим МП–автомат

$$M = (K, V_t, \Gamma, \delta, p_0, F, B_0)\ ,$$

где $\Gamma = V_t \cup V_n \cup \{B_0\}, K = \{p_0, f\}, F = \{f\}$ и функция переходов δ содержит команды следующего вида:

a) $p_0, a, \varepsilon \to p_0, a$ для любого терминала $a \in V_t$;

b) $p_0, \varepsilon, \widetilde{\varphi} \to p_0, A$ для всех правил $A \to \varphi \in P$ ($\widetilde{\varphi}$ — зеркальное отображение φ) ;

c) $p_0, \varepsilon, SB_0 \to f, B_0$.

Очевидно, что любому выводу в G взаимно однозначно соответствует последовательность команд построенного автомата M.

Обратное построение КС–грамматики по произвольному МП–автомату также возможно, но не представляет практического интереса. □

Пример 2.10. Построим МП–автомат для грамматики

$$G: \begin{array}{l} S \to S + A | S - A | A \\ A \to a | (S) \end{array}$$

Эквивалентный МП–автомат содержит следующие команды:

a) команды переноса терминалов в магазин

$$\begin{array}{ll} P_0, a, \varepsilon & \to P_0, a \\ P_0, +, \varepsilon & \to P_0, + \\ P_0, -, \varepsilon & \to P_0, - \\ P_0, (, \varepsilon & \to P_0, (\\ P_0,), \varepsilon & \to P_0,) \end{array}$$

b) команды редукции по правилам грамматики

$$\begin{array}{ll} P_0, \varepsilon, A + S & \to P_0, S \\ P_0, \varepsilon, A - S & \to P_0, S \\ P_0, \varepsilon,)S(& \to P_0, A \\ P_0, \varepsilon, A & \to P_0, S \\ P_0, \varepsilon, a & \to f, A \end{array}$$

c) команду проверки на завершение

$$P_0, \varepsilon, SB_0 \to f, B_0$$

Подадим на вход МП–автомата цепочку $(a+a)-a$. Действия, которые выполняет этот автомат, представлены на рис. 2.13.

На третьем шаге можно применить одну из двух команд: $P_0, \varepsilon, a \to P_0, A$ или $P_0, +, \varepsilon \to P_0, +$. Применение второй из них не приведет к завершению разбора.

Рассмотренное доказательство теоремы 2.11 основано на восходящей стратегии разбора. Рассмотрим нисходящую стратегию разбора. На каждом шаге нисходящей стратегии разбора должно применяться какое–либо правило. В начальный момент таким нетерминалом является аксиома. Тогда соответствующий МП–автомат должен выполнять следующие действия:

состояние	магазин	вход
P_0	B_0	(
P_0	B_0 (	a
P_0	B_0 (a	
P_0	B_0 (A	
P_0	B_0 (S	+
P_0	B_0 (S +	a
P_0	B_0 (S + a	
P_0	B_0 (S + A	
P_0	B_0 (S	)
P_0	B_0 (S)	
P_0	B_0 A	
P_0	B_0 S	-
P_0	B_0 S -	a
P_0	B_0 S - a	
P_0	B_0 S - A	
P_0	B_0 S	
F	B_0	

Рис. 2.13: Грамматический разбор цепочки $(a + a) - a$ по восходящей стратегии для грамматики примера 2.10.

a) в начальный момент в магазин заносится аксиома:

$$p_0, \varepsilon, \varepsilon \to p_1, S;$$

b) для любого правила $A \to \varphi \in P$ нетерминал заменяется на правую часть правила с помощью команды

$$p_1, \varepsilon, A \to p_1, \widetilde{\varphi};$$

c) для любого терминала $a \in V_t$ выполняется сравнение символа на входе с символом в верхушке магазина по команде

$$p_1, a, a \to p_1, \varepsilon$$

(символ a был записан в магазин на некотором предшествующем шаге разбора, когда применялась правило грамматики);

d) разбор заканчивается по команде

$$p_1, \varepsilon, B_0 \to f, B_0,$$

когда цепочка прочитана полностью и магазин пуст.

Пример 2.11. Для грамматики из примера 2.10 работающий по нисходящей стратегии МП–автомат имеет множество команд:

a) команда записи аксиомы в магазин в начале работы

$$P_0, \varepsilon, \varepsilon \to P_1, S;$$

б) команды применения правил грамматики к нетерминалу в верхушке магазина

$$\begin{array}{l} P_1, \varepsilon, S \to P_1, A+S \\ P_1, \varepsilon, S \to P_1, A-S \\ P_1, \varepsilon, S \to P_1, A \\ P_1, \varepsilon, S \to P_1, A \\ P_1, \varepsilon, A \to P_1,)S(\\ P_1, \varepsilon, A \to P_1, a; \end{array}$$

в) команды сравнения терминала в верхушке магазина с терминалом на входной ленте

$$\begin{array}{l} P_1, a, a \to P_1, a \\ P_1, +, + \to P_1, \varepsilon \\ P_1, -, - \to P_1, \varepsilon \\ P_1, (, (\to P_1, \varepsilon \\ P_1,),) \to P_1, \varepsilon; \end{array}$$

г) команда завершения работы

$$P_1, \varepsilon, B_0 \to F, B_0.$$

Подадим на вход цепочку $(a + a) - a$, тогда процесс разбора может быть представлен на рис. 2.14. Сразу отметим, что при нисходящей стратегии разбора недетерминированный МП–автомат может выполнить грамматический разбор в леворекурсивной КС–грамматике именно в силу своей недетерминированности. Детерминированная модель недетерминированного алгоритма, построенная на основе перебора всех вариантов применения подходящего правила, не может работать в леворекурсивной КС–грамматике. Дело в том, что если при нисходящем грамматическом разборе анализатор по некоторым причинам решил применить леворекурсивное правило $A \to A\beta$, то правая часть $A\beta$ этого правила заменит в магазине находящийся там нетерминал A, причем в верхушке магазина опять окажется тот же самый нетерминал A, а, значит, ситуация повторяется. В результате анализатор бесконечно будет применять одно и то же леворекурсивное правило.

2.9 Разбор с возвратом

Рассмотренные МП–автоматы работают недетерминированно. Если цепочка принадлежит языку, порождаемому грамматикой, то какой–то из вариантов функционирования автомата выполнит правильный разбор. Если цепочка не принадлежит языку, то никакой вариант не приведет к цели. Отсутствие эквивалентного детерминированного автомата для произвольной КС–грамматики означает невозможность построения универсальной простой однопроходной программы синтаксического анализа. Поэтому для эффективного разбора необходимо выделять специальные классы КС–грамматик, удовлетворяющие требованиям конкретных типов анализаторов. Если требуется выполнить разбор для произвольной КС–грамматики, то придется использовать детерминированную программную модель недетерминированного МП–автомата.

Для того, чтобы запрограммировать недетерминированный МП–автомат, необходимо обеспечить перебор всех возможных вариантов на каждом шаге разбора. В общем виде для любой стратегии перебора алгоритм реализуется рекурсивной процедурой и должен обеспечивать следующие действия.

состояние	магазин	вход
P_0	B_0	
P_1	B_0 S	
P_1	B_0 A - S	
P_1	B_0 A - A	
P_1	B_0 A -) S (	(
P_1	B_0 A -) S	
P_1	B_0 A -) S $+$ A	
P_1	B_0 A -) S $+$ a	a
P_1	B_0 A -) S $+$	$+$
P_1	B_0 A -) S	
P_1	B_0 A -) A	
P_1	B_0 A -) a	a
P_1	B_0 A -)	)
P_1	B_0 A -	-
P_1	B_0 A	
P_1	B_0 a	a
P_1	B_0	
F	B_0	

Рис. 2.14: Грамматический разбор цепочки $(a + a) - a$ по нисходящей стратегии.

1) Проверяется условие завершения разбора. Для нисходящей стратегии условием успешного завершения разбора является пустой магазин и полностью прочитанная цепочка, для восходящей — в магазине находится аксиома и цепочка прочитана полностью. Если разбор закончен, то рекурсивная функция грамматического разбора возвращает значение 1 в качестве признака успешного разбора и завершает работу.

2) Проверяется условие невозможности дальнейшего разбора. Для нисходящей стратегии разбора таким условием является несовпадение терминального символа в верхушке магазина с терминалом на входе. Момент возврата для разбора по восходящей стратегии определяется по условию невозможности продвижения вперед по исходной цепочке: цепочка прочитана полностью, а дерево построено лишь частично или не построено вообще, т.е. в магазине находится не аксиома грамматики.

3) Если оба предшествующих условия не выполнены, это означает, что разбор можно продолжить, применяя правила грамматики. Поэтому организуется цикл по всем правилам грамматики, предусматривающий выполнение следующей последовательности действий.

a) Применяется очередное правило грамматики.

b) Выполняется рекурсивный вызов функции грамматического разбора.

c) Проверяется признак возврата из рекурсивной функции. Если функция вернула 1, то действия были выполнены правильно и разбор успешно завершен. Если функция вернула 0, то текущее правило было неверным. Тогда восстанавливается состояние до применения правила — это верхушка магазина и указатель входной цепочки. После восстановления переходим к следующему правилу грамматики, если оно существует. При завершении перебора, когда ни одно правило не привело к успешному завершению разбора, функция возвращает 0.

Приведем в качестве примера универсальную программу нисходящего разбора с возвратом. Разбор выполняет рекурсивная функция $Rec(int\ k)$, параметр которой определяет положение указателя исходной цепочки. Пусть правила грамматики задаются в файле по строкам. Каждая строка определяет одно правило грамматики, причем нетерминалами являются большие латинские буквы, а терминалами — маленькие латинские буквы. Нетерминал в левой части правила отделяется от правой части последовательностью символов $->$, окруженной пробелами.

Сразу отметим, что рекурсивный алгоритм нисходящего разбора не может работать на леворекурсивных правилах, т.к. выбор первого леворекурсивного правила грамматики блокирует все остальные. Единственный результат, который будет достигнут в этом случае — сообщение *stack overflow!* . Это ограничение распространяется на любые нисходящие методы разбора, а не только на рекурсивные.

Левая рекурсия может быть не только явной, но и скрытой, например, при наличии правил $S \to Ac$, $A \to BaB$, $B \to SaA$ получаем леворекурсивную зависимость $S \Rightarrow Ac \Rightarrow BaBc \Rightarrow SaAaBc$. Скрытая рекурсия при нисходящем разборе также приведет к сообщению о переполнении стека.

```
// грамматический разбор на нисходящий стратегии с возвратами
#include <stdio.h>
#include <string.h>
#include <STDLIB.H>

#define  MAX_STACK 2000
#define  MAXK  100    // максимальное число правил грамматики
#define  MAX_LEX 500  // максимальная длина исходной цепочки

typedef char LEX[MAX_LEX];
int len[MAXK];
char a[MAXK]; //левые части
LEX fi[MAXK]; // правые части правил грамматики
int  m;       // фактическое число правил грамматики
LEX x;        // исходная цепочка
FILE *gr = fopen("gramm.txt","r"); //файл, содержащий правила грамматики
FILE *in = fopen("input.txt","r"); // файл с исходной цепочкой
char stack[MAX_STACK];    //стек
int uk;                   // указатель первого свободного элемента стека

void GetData(void)
{
LEX ss; // вспомогательная строка для ввода "->"
int i;
fscanf(gr,"%d\n",&m);// число правил
for (i=0; i<m; i++)
     {
      fscanf(gr,"%c %s %s\n",&a[i],&ss,&fi[i]);
       // правило задается в форме a -> fi
     if (fi[i][0]=='e') {len[i]=0; fi[i][0]='\0';}
         else len[i]=strlen(fi[i]);
     }
fscanf(in,"%s",&x);
```

```
}

int Rec(int k)
// грамматический разбор от текущей позиции
{
int i,j,old_uk;
if ( (x[k]=='\0') && (uk==0)) return 1;
if ( (uk>0) && (stack[uk-1]<='z') && (stack[uk-1]>='a'))
    { // в верхушке магазина терминал
    if (x[k]==stack[uk-1]) { uk--; return Rec(k+1);}
       else return 0;
    }
// в верхушке магазина нетерминал
for (i=0; i<m; i++)
    if (stack[uk-1]==a[i])
       { // применяем правило грамматики
       old_uk=uk; uk--;
       printf("Правило %c -> %s   указатель цепочки k=%d\n",
               a[i], fi[i],k);
       for (j=0; j<len[i]; j++)
            stack[uk-j+len[i]-1]=fi[i][j]; // зеркальное отображение
       uk+=len[i];
       if (Rec(k)==1)
         { printf ("Правильно!\n");
         return 1;
         }
       // возврат:
       uk=old_uk; stack[uk-1] = a[i]; printf("Возврат!\n");
       }
return 0;
}


int main(void)
{
GetData();
uk=1; stack[0]=a[0]; // первый нетерминал - аксиома
if (Rec(0)==1) printf("Разбор закончен.\n");
    else printf("Ошибки во входной цепочке.\n");
fclose(in); fclose(gr);
return 0;
}
```

Пример файла gram.txt:
2
$S -> aSb$
$S -> c$
Пример файла input.txt для правильной цепочки:
$aaacbbb$
Заметим, что практического значения при построении трансляторов рассмотрен-

ный алгоритм не имеет, т.к. он основан на переборе всех применимых на текущем шаге действий. Используемые на практике КС–грамматики языков программирования удовлетворяют дополнительным ограничениям, которые позволяют строить более эффективные алгоритмы синтаксического анализа. Как правило, КС–грамматики языков программирования обладают такими свойствами, что существуют алгоритмы грамматического разбора с временной сложностью порядка $O(|x|)$, где $|x|$ — длина анализируемой цепочки.

Тем не менее алгоритм разбора с возвратом представляет не только теоретический, но и практический интерес, т.к. *общая схема нисходящего синтаксического анализа произвольной КС–грамматики, основанная на замене нетерминала в верхушке магазина правой частью правила, остается неизменной и для алгоритмов эффективного разбора.*

В качестве упражнения оставляется задача реализации универсального рекурсивного восходящего анализатора — программы грамматического разбора с возвратом по восходящей стратегии разбора. Замечание относительно эффективности грамматического разбора по нисходящей стратегии с возвратом полностью справедливо и для восходящей стратегии разбора.

2.10 Контрольные вопросы к разделу

1. Что представляет собой память автомата?

2. Приведите иерархию автоматов по сложности .

3. Чем отличаются автоматы — преобразователи от автоматов — распознавателей?

4. Перечислением каких объектов задается конечный автомат?

5. Какое множество называется регулярным?

6. Какие операции не выводят из класса регулярных множеств?

7. Как построить автомат, распознающий объединение регулярных множеств?

8. Почему при доказательстве теоремы об объединении двух регулярных языков необходимым условием является отсутствие циклов в начальных состояниях конечных автоматов, распознающих эти языки?

9. Почему при доказательстве теоремы о дополнении регулярного языка необходимым условием являетcя детерминированность конечного автомата?

10. Как построить автомат, распознающий дополнение регулярного множества?

11. Какие два состояния конечного автомата называются k–эквивалентными?

12. Почему при доказательстве теоремы о минимизации конечного автомата на первом шаге все заключительные состояния отделяются от незаключительных?

13. Дайте определение автомата с магазинной памятью.

14. Как для заданной грамматики построить МП–автомат, выполняющий восходящий разбор?

15. Как для заданной грамматики построить МП–автомат, выполняющий нисходящий разбор?

16. Почему МП–автомат, построенный для произвольной КС–грамматики, является в общем случае недетерминированным?

17. Как написать программу, выполняющую восходящий грамматический разбор в соответствии с командами недетерминированного МП–автомата?

18. Как написать программу, выполняющую нисходящий грамматический разбор в соответствии с командами недетерминированного МП–автомата?

19. Чему равна верхняя граница числа шагов при минимизации произвольного конечного автомата?

20. Чем МП–автомат отличается от конечного автомата?

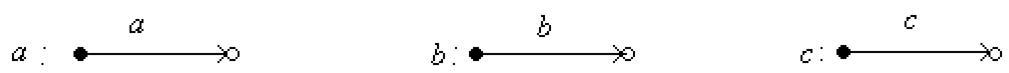
2.11 Упражнения к разделу

Задание. Построить детерминированный конечный автомат, распознающий заданный язык L. Для полученного автомата построить эквивалентую леволинейную и эквивалентную праволинейную грамматику. Привести пример цепочки $x \in L$, показать процесс распознавания автоматом этой цепочки, а также построить вывод этой цепочки в обеих грамматиках.

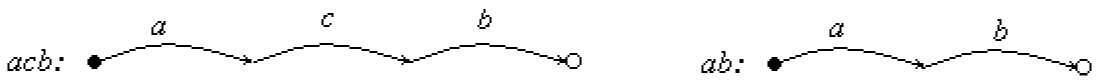
2.11.1 Задача

$$L = (ab)^*c^* \cup (acb)^* \cup a^+.$$

Решение. Очевидно, что элементарные языки, содержащие по одной цепочке a, b, c соответственно, распознаются автоматами, представленными на рисунке:



С помощью операции произведения получим языки ab, acb, а для построения соответствующих автоматов просто соединим начальное состояние автомата b с заключительным состоянием автомата a — для языка ab, и последовательно заключительное состояние автомата a с начальным состоянием автомата c и заключительное состояние этого автомата с начальным состоянием автомата b — для языка acb. Построенные автоматы имеют вид:



Итерация языка распознается автоматом, у которого объединены начальное и заключительное состояние исходного автомата. Соответствующие автоматы представлены на рис. 8.15.

Рассмотрим теперь построение автомата, распознающего язык $(ab)^*c^*$ — произведение языков $(ab)^*$ и c^*. В процессе доказательства теоремы об операциях над регулярными языками мы выяснили, что при построении автоматов, распознающих произведение двух заданных языков необходимо избавиться от циклов в одном из объединяемых состояний — заключительном состоянии первого автомата или в начальном состоянии второго автомата. Удалим циклы из начального состояния автомата, распознающего c^*, а затем объединим указанные состояния. В результате получим автоматы, представленные на рис. 8.16.

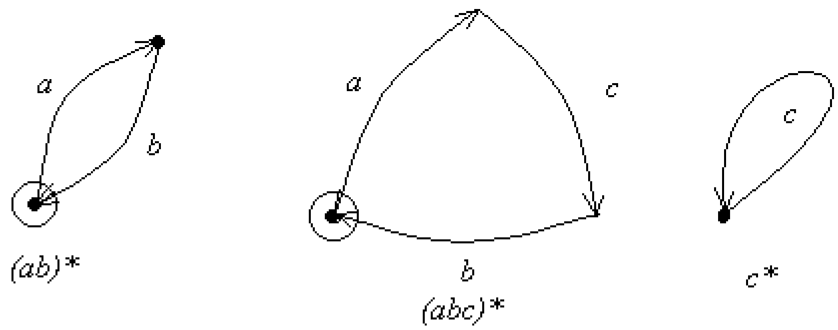


Рис. 2.15: Конечные автоматы, распознающие итерации языков ab, acb, c.

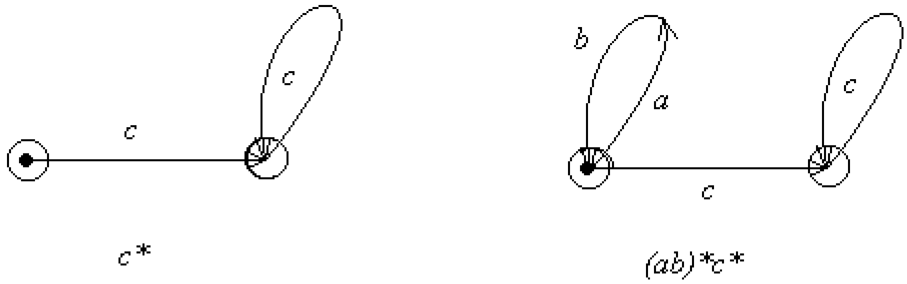


Рис. 2.16: Конечные автоматы, распознающие языки c^*, $(ab)^*c^*$.

Теперь для построения автомата, распознающего заданный язык

$$(ab)^*c^* \cup (acb)^* \cup a^+.$$

достаточно воспользоваться той же теоремой и объединить начальные состояния автоматов $(ab)^*c^*$, $(acb)^*$ и a^+, предварительно устранив циклы из начальных состояний, если они там были. Соответствующие автоматы и результирующий автомат представлены на рис. 8.17.

Построенный автомат не является детерминированным, поэтому необходимо выполнить алгоритм детерминизации полученного автомата. Сначала построим таблицу переходов полученного автомата:

	a	b	c	
p_0	$p_1p_4p_5$		p_3	закл
p_1		p_2		
p_2	p_1		p_3	закл
p_3			p_3	закл
p_4	p_4			закл
p_5			p_7	
p_6	p_5			закл
p_7		p_6		

Начиная из состояния p_0 будем строить объединенные состояния нового автомата:

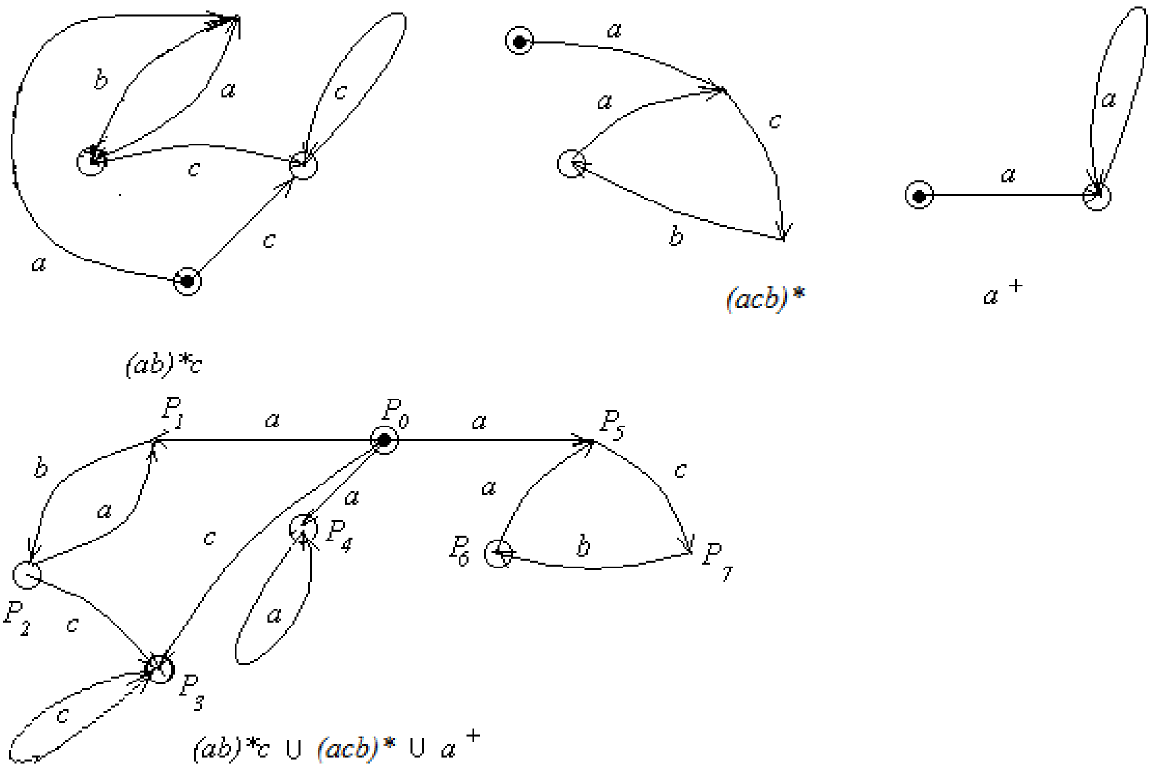


Рис. 2.17: Конечные автоматы без циклов в начальных состояниях, распознающие языки $(ab)^*c^*$, $(acb)^*$ и a^+; автомат, распознающий язык $(ab)^*c^* \cup (acb)^* \cup a^+$.

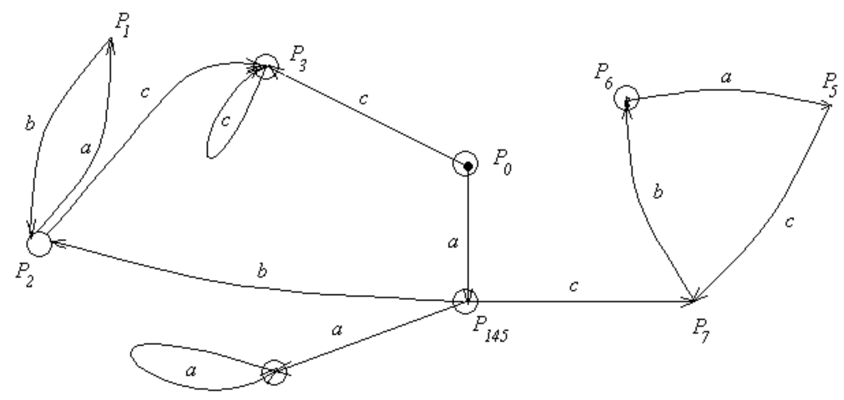


Рис. 2.18: Детерминированный конечный автомат, распознающий язык $(ab)^c \cup (acb)^* \cup a^+$.

	a	b	c	
p_0	p_{145}		p_3	закл
p_{145}	p_4	p_2	p_7	закл
p_4	p_4			закл
p_2	p_1		p_3	закл
p_7		p_6		
p_1		p_2		
p_3			p_3	закл
p_6	p_5			закл
p_5			p_7	

Следует отметить, что у исходного автомата недетерминированным являлось только одно состояние p_0, а второй символ цепочки однозначно определяет тип цепочки, поэтому только одно новое состояние p_{145} появляется у нового детерминированного автомата. Построенный автомат представлен на рис. 8.17.

Построим теперь праволинейную и леволинейную грамматики, эквивалентные полученному детерминированному автомату. Для этого сначала каждому состоянию автомата поставим в соответствие нетерминальный символ:

состояние	p_0	p_{145}	p_4	p_2	p_7	p_1	p_3	p_6	p_5
нетерминал	N	B	C	D	E	F	K	T	M

Для праволинейной грамматики аксиомой является символ N, соответствующий начальному состоянию автомата. Для каждой команды автомата запишем правило грамматики. Например, команде $p_2, a \to p_1$ ставится в соответствие правило $D \to aF$. Добавим терминальные правила, определяющие завершение процесса порождения цепочки в грамматике. Например, правило $F \to b$ появляется из-за команды перехода $p_1, b \to p_2$ в заключительное состояние p_2. Остальные правила строятся аналогично. Получим праволинейную грамматику:

$$
\begin{aligned}
G_1: \quad & N \to aB|a|cK|c \\
& B \to aC|bD|cE|a|b \\
& C \to aC|a \\
& D \to aF|cK|c \\
& E \to bT|b \\
& F \to bD|b \\
& K \to cK|c \\
& T \to aM \\
& M \to cE
\end{aligned}
$$

Перейдем теперь к построению леволинейной грамматики. В качестве аксиомы возьмем дополнительный нетерминальный символ S, для которого в множество правил грамматики необходимо включить правила, соответствующие командам автомата перехода в заключительное состояние. Например, для команды $p_{154}, b \to p_2$ перехода в заключительное состояние p_2 записываем правила грамматики $S \to Bb$. Терминальные правила грамматики строятся для команд перехода из начального исходного состояния автомата. Например, правило $B \to a$ записывается для команды $p_0, a \to p_{145}$. В результате таких построений получим грамматику:

$$
\begin{aligned}
G_2: \quad & S \to Bb|Ca|Ba|Na|Eb|Db|Kc \\
& N \to \\
& B \to Na|a \\
& C \to Ba|Ca \\
& D \to Bb|Fb \\
& E \to Bc|Mc \\
& F \to Da \\
& K \to Kc|Nc|Dc|c \\
& T \to Eb \\
& M \to Ta
\end{aligned}
$$

Построенная грамматика содержит непродуктивный нетерминал N. Приведем грамматику:

$$
\begin{aligned}
G_3: \quad & S \to Bb|Ca|Ba|Eb|Db|Kc \\
& B \to a \\
& C \to Ba|Ca \\
& D \to Bb|Fb \\
& E \to Bc|Mc \\
& F \to Da \\
& K \to Kc|Nc|Dc|c \\
& T \to Eb \\
& M \to Ta
\end{aligned}
$$

Следует заметить, что подстановка терминального правила $B \to a$ в правые части правил грамматики G_3 уменьшит число нетерминалов, но в результате нарушится структура правил и грамматика перестанет быть леволинейной.

Наконец, рассмотрим примеры вывода в грамматиках и процессы распознавания цепочек исходным автоматом. Заданный язык

$$(ab)^*c^* \cup (acb)^* \cup a^+.$$

содержит цепочки трех типов. Для контроля правильности наших построений рассмотрим примеры цепочек каждого из указанных типов.

а) Цепочка aaa распознается автоматом с помощью команд:

$$p_0, a \to p_{154},\ p_{154}, a \to p_4,\ p_4, a \to p_4.$$

Эта же цепочка порождается в грамматике G_1:

$$N \Rightarrow aB \Rightarrow aaC \Rightarrow aaa.$$

Вывод aaa в грамматике G_3:

$$S \Rightarrow Ca \Rightarrow Baa \Rightarrow aaa.$$

б) Цепочка $acbacb$ распознается автоматом:

$$p_0, a \to p_{154},\ p_{154}, c \to p_7,\ p_7, b \to p_6,\ p_6, a \to p_5,\ p_5, c \to p_7,\ p_7, b \to p_6.$$

Эта же цепочка порождается в грамматике G_1:

$$N \Rightarrow aB \Rightarrow acE \Rightarrow acbT \Rightarrow acbaM \Rightarrow acbacE \Rightarrow acbacb.$$

Вывод $acbacb$ в грамматике G_3:

$$S \Rightarrow Eb \Rightarrow Mcb \Rightarrow Tacb \Rightarrow Ebacb \Rightarrow Bcacb \Rightarrow acbacb.$$

в) Цепочка $ababcc$ распознается автоматом:
$p_0, a \to p_{154}$, $p_{154}, b \to p_2$, $p_2, a \to p_1$, $p_1, b \to p_2$, $p_2, c \to p_3$, $p_3, c \to p_3$.
Эта же цепочка порождается в грамматике G_1:

$$N \Rightarrow aB \Rightarrow abD \Rightarrow abaF \Rightarrow ababD \Rightarrow ababcK \Rightarrow ababcc.$$

Вывод $ababcc$ в грамматике G_3:

$$S \Rightarrow Kc \Rightarrow Dcc \Rightarrow Fbcc \Rightarrow Dabcc \Rightarrow Bbabcc \Rightarrow ababcc.$$

2.11.2 Варианты заданий

1. $(cab)^+ \cup (b)^* \cup bc^*$.
2. $(aca)^* \cup (ca)^*cb^* \cup ac^*$.
3. Дополнение $bac^* \cup (bc)^+$.
4. $ac^* \cup (bca)^*(ba)^*$.
5. Дополнение $bac^* \cup (ac)^+$.
6. $(baa)^*(bc)^* \cup ca^*ac^*$.
7. $(ba)^*c^* \cap (a^*c^*b^*)^+$.
8. $(ca)^+ \cup (b)^* \cup bc^* \cup cc \cup ac$.
9. Дополнение $b^*a \cup bab^+$.
10. $ac^*a \cup (bca)^* \cup (ba)^*$.
11. $cc(ba)^+ \cup (ca^*ac)^* \cup a^*$.
12. $(ab)^*(a^*ba)^* \cup (bac)^*$.
13. $ac^*(ba)^* \cup (ca)^*cb^*$.
14. $(ab)^*(cc)^* \cap (b^*c^*a^*)^*$.
15. Дополнение $ac^* \cup (bca)^*$.
16. $(ba)^+ \cup ca^*ac^* \cup a^*$.
17. $(bc)^* \cup bc^*a \cup cc^+ \cup ac^*$.
18. $ac^*(bca)^* \cup (ba)^*ca^*$.
19. Дополнение $c^*a \cup (cac)^+$.
20. $(baa)^* \cup (bc)^* \cup ca^* \cup ac^*$.

2.12 Тесты для самоконтроля к разделу

1. Автоматом какого типа распознается язык $a^{n+1}b * c^n \cup (ab)^*$? Если существуют несколько типов автоматов, распознающих данный язык, укажите наиболее простой из них.

Варианты ответов:
а) недетерминированным конечным автоматом;
б) детерминированным конечным автоматом;
в) недетерминированным МП–автоматом;
г) детерминированным МП–автоматом;
д) недетермирированной машиной Тьюринга;
е) детерминированной машиной Тьюринга.

Правильный ответ: в.

2. Укажите ложные утверждения из следующего перечня утверждений.

1) Для любого недетерминированного конечного автомата можно построить эквивалентный детерминированный конечный автомат.

2) Для любого недетерминированного МП–автомата можно построить эквивалентный детерминированный МП–автомат.

3) Для любого недетерминированного конечного автомата можно построить эквивалентный детерминированный МП-автомат.

Варианты ответов:
а) ложно 1;
б) ложно 2;
в) ложно 3;
г) ложно 1 и 2;
д) ложно 1 и 3;
е) ложно 2 и 3;
ж) ложно 1, 2 и 3.

Правильный ответ: б.

3. Сколько состояний содержит минимальный конечный автомат, распознающий язык $a^*b^*a^* \cup a^*b^+c^*$?

Варианты ответов:
а) 1;
б) 2;
в) 3;
г) 4;
д) 5;
е) 6.

Правильный ответ: г.

4. Какие из следующих утверждений истинны?

1) Пересечение произвольных регулярных языков является регулярным.

2) Пересечение произвольных КС–языков является КС–языком.

3) Пересечение произвольного регулярного языка и произвольного КС–языка является КС–языклм.

Варианты ответов:
а) все утверждения ложны;
б) истинно только 1;

в) истинно только 2;
г) истинно только 3;
д) истинны 1 и 2;
е) истинны 1 и 3;
ж) истинны 2 и 3;
з) все утверждения истинны.

Правильный ответ: е.

5. Необходимо построить МП–автомат, выполняющий восходящий грамматический разбор в грамматике

$$G : S \longrightarrow aSbb|a$$

Какое множество команд должен содержать автомат, выполняющий указанные действия?

Варианты ответов:

$$\text{а)}\quad \begin{array}{l} p_0, \varepsilon, S \longrightarrow p_0, a \\ p_0, \varepsilon, S \longrightarrow p_0, bbSa \\ p_0, a, a \longrightarrow p_0, \varepsilon \\ p_0, b, b \longrightarrow p_0, \varepsilon \\ p_0, \varepsilon, B_0 \longrightarrow p_1, B_0 \end{array}$$

$$\text{б)}\quad \begin{array}{l} p_0, a \longrightarrow p_0 \\ p_0, b \longrightarrow p_1 \\ p_1, b \longrightarrow p_1 \\ p_1, \varepsilon \longrightarrow p_2 \end{array}$$

$$\text{в)}\quad \begin{array}{l} p_0, a, a \longrightarrow p_0, \varepsilon \\ p_0, b, b \longrightarrow p_0, \varepsilon \\ p_0, \varepsilon, bbSa \longrightarrow p_0, S \\ p_0, \varepsilon, a \longrightarrow p_0, S \\ p_0, \varepsilon, S \longrightarrow p_1, \varepsilon \end{array}$$

$$\text{г)}\quad \begin{array}{l} p_0, a, \varepsilon \longrightarrow p_0, a \\ p_0, b, \varepsilon \longrightarrow p_0, b \\ p_0, \varepsilon, S \longrightarrow p_0, bbSa \\ p_0, \varepsilon, S \longrightarrow p_0, a \\ p_0, \varepsilon, S \longrightarrow p_1, \varepsilon \end{array}$$

$$\text{д)}\quad \begin{array}{l} p_0, a, \varepsilon \longrightarrow p_0, a \\ p_0, b, \varepsilon \longrightarrow p_0, b \\ p_0, \varepsilon, bbSa \longrightarrow p_0, S \\ p_0, \varepsilon, a \longrightarrow p_0, S \\ p_0, \varepsilon, S \longrightarrow p_1, \varepsilon \end{array}$$

Правильный ответ: д.

Глава 3

ЛЕКСИКА, СИНТАКСИС И СЕМАНТИКА ЯЗЫКА

3.1 Понятие языка программирования и языкового процессора

Под термином "язык программирования" понимают как языки высокого уровня, проблемно–ориентированные языки, так и бедно структурированные машино–ориентированные или машинные языки. Машинный язык представляет собой совокупность шестнадцатиричных цифр и практически не используется программистами. Обычно для низкоуровневого программирования используется язык Ассемблера. Будем исходить из того, что программа написана на каком–либо языке и представляет собой последовательность символов, которые программист сумел набрать с клавиатуры. Такая программа называется исходным модулем. *Компилятор* рассматривает исходный модуль в качестве своих исходных данных и преобразует ее в программу на объектном машино–ориентированном языке. При этом естественно предполагается, что сгенерированная компилятором программа выполняет те же действия, что и предусматривал программист при написании исходной программы. При этом вовсе не обязательно, чтобы сгенерированная программа выполнялась на том же самом компьютере или компьютере того же типа, на котором работает компилятор. *Интерпретатор* языка получает на вход программу на исходном языке и выполняет ее на имеющемся компьютере. Таким образом, интерпретатор, в отличие от компилятора, не вырабатывает какую-либо программу в качестве результата, а выполняет действия, заданные в исходной программе. Программа интерпретатора для небольших языков существенно проще программы компилятора. Принцип интерпретации имеет еще одно дополнительное преимущество, которое состоит в относительной независимости от ЭВМ. Все программное обеспечение, написанное на интерпретируемом языке программирования, без изменения переносится на другую машину, изменению подлежит лишь программа самого интерпретатора.

Компилятор и интерпретатор обычно являются довольно сложными программами, которые получают на вход исходный модуль в форме текста, устанавливают его внутреннюю структуру, проверяя при этом его синтаксическую корректность и выполняя некоторый семантический контроль.

Любой *языковый процессор* — это программа, получающая на вход произвольную цепочку x некоторого языка L_1 и преобразующая ее в цепочку y некоторого другого языка L_2 по определенному закону:

$$y = F(x). \tag{3.1}$$

Одним из самых распространенных примеров такого отображения является компиляция исходного модуля с языка программирования высокого уровня (например, *.cpp) в объектный код (*.obj). Другим примером является интерпретация файла *.bat в среде MS DOS, в результате которой выполняется определенная последовательность действий.

Ограничимся пока проблемами, связанными с отображением одного представления алгоритма в другое его представление. Рассматривая языки программирования в качестве инструмента для описания алгоритмов, будем говорить о процессе трансляции языка в соответствии с законом (3.1). В зависимости от вида функции отображения (3.1) различают следующие виды трансляторов.

1) *Ассемблеры* - это трансляторы, работающие по принципу "один в один", т.е. один оператор языка программирования транслируется в один оператор результирующего языка (например, одна команда входного языка TASM в одну команду объектного кода).

2) *Макроассемблеры* — трансляторы, работающие по принцину "один в несколько", когда транслируемые языки допускают использование макрокоманд, не имеющих прямых аналогов в машинном эквиваленте или другом результирующем представлении.

3) *Компиляторы* — это трансляторы, работающие по принципу "несколько в несколько". Обычно компиляторы используются для трансляции так называемых языков высокого уровня (Java, C++, C♮, Паскаль и т.п.). Языки высокого уровня содержат средства описания данных и их типов, а также различные типы структурных операторов программы. Например, при трансляции конструкции в скобках оператора for языка C++

```
for (int index = 1; index < num; index++) s += fun(index);
```

требуется сформировать команду увеличения на 1 значения переменной $index$, имя которой указано в начале оператора цикла, а также организовать переход на проверку условия $index \leq num$, при этом проверяется, что идентификаторы num, s и fun объявлены и имеют соответствующие типы.

4) *Генераторы* — это трансляторы, позволяющие выполнять генерацию программы. Современным типом таких трансляторов являются *графовые компиляторы* , которые транслируют не исходный код программы, а модель задачи, представленную в виде графа. Если алгоритмы, применяемые в некоторой конкретной области хорошо известны, методы расчетов соответствуют элементам модели (например, электронным схемам), то по модели устройства графовый компилятор может сгенерировать программу расчета прибора в целом. Генератор формирует программу, поэтому основная трудность на пути создания генераторов заключается в построении интерактивного взаимодействия с пользователем и в создании программ автоматического выбора методов генерации. Работает генератор по принципу "несколько в очень много". В нашем курсе мы не будем рассматривать генераторы.

Основной предмет нашего курса — компиляторы, т.е. трансляторы с языков высокого уровня. Рассмотрим некоторые типы компиляторов, существующие в настоящее время. Прежде всего, отметим, что в классическом понимании компилятор транслирует исходный код в объектный код. В современной терминологии существуют и другие типы трансляторов.

• *Интерпретаторы* не генерируют объектный код, а исполняют его. Интерпретация может осуществляться либо непосредственно на уровне исходного кода (например, браузер интерпретирует код на языке JavaScript в html–документе), либо на уровне промежуточного кода, полученного в результате трансляции (например, Java-машина интерпретирует байт–код). Можно привести много примеров интерпретаторов: операционная система интерпретирует командный файл, MS Word интерпретирует файл с текстом в формате rtf, интерпретируется программа на языке Prolog, и т.п.

• *Кросс–компиляторы* — компиляторы, работающие на одной платформе и генерирующие код для другой платформы. Такой инструмент бывает полезен, когда нужно получить код для платформы, экземпляров которой нет в наличии, или в случаях, когда компиляция на целевой платформе невозможна или нецелесообразна (например, это касается мобильных систем или микроконтроллеров с минимальным объҷмом памяти). Пример кросс–компилятора — GCC, который может быть установлен как кросс-компилятор. с опцией -mno-cygwin. С этой опцией он может в среде Cygwin создавать код, использующий библиотеки Windows. Компания Manx Software Systems производит кросс–компиляторы для разных платформ, включая PC и Macs. Компилятор командной строки с языка C в Visual Studio может генерировать нативный код для разных процесоров. Позволяет выполнять кросс-компиляции Lazarus и т.п.

• *Параллельные компиляторы* — новое направление в построении компиляторов, направленное на создание кода, выполняющегося на параллельной архитектуре с многоядерными процессорами. Большую работу в этом направлении проводит, например, фирма Intel.

• *Динамические компиляторы* — компиляторы, которые вызываются во время выполнения программы в промежуточном коде для ее трансляции в машинный код. Такие компиляторы используются, например, при выполнении на конкретной платформе байт-кода, полученного при трансляции программы на языке Java. При первом вызове каждого метода некоторого класса байт-код транслируется в машинный код, который затем будет выполняться при каждом последующем вызове. Такие компиляторы получили название динамических (*Just-in-time или JIT*) компиляторов отличие от предкомпиляторов (*Ahead-of-time или AOT*), которые выполняют аналогичную трансляцию байт-кода в машинный код перед выполнением программы, а не во время ее выполнения.

• *Конверторы* транслируют программу с одного языка высокого уровня на другой. Это достаточно редкий тип транслятора, который используется, как правило, для переноса старого программного кода уже существующих и работающих систем на новый язык программирования.

При задании языков программирования описывается синтаксис языка с помощью грамматических правил. Для выполнения грамматического разбора языковый процессор должен содержать *синтаксический анализатор*. Из теории формальных грамматик и языков известно, что КС–грамматики, которые обычно применяются для описания синтаксиса языков программирования, не отражают некоторые правила построения правильных программ на этих языках. Это обстоятельство вызвано тем, что языки программирования не являются контекстно–свободными и имеют более сложную по сравнению с последними синтаксическую структуру. Синтаксические правила языков программирования, которые не описываются средствами КС–грамматик и часто задаются неформально, называются *контекстными условиями*. Программу, удовлетворяющую контекстным условиям, будем называть семантически правильной.

Важная задача, решаемая при переводе с языка на язык — это обеспечение совпадения смысла исходной программы и результата перевода. Очевидно, что необходимым условием перевода с языка L_1 на язык L_2 является включение смыслового множества языка L_1 в смысловое множество языка L_2. Например, перевод с языка Оккам, предназначенного для описания параллельных алгоритмов, вряд ли возможен на язык C++, если только для целей такого перевода не реализованы на C++ специальные классы выполнения параллельных процессов. Необходимость отображения смысла при переводе (3.1) реализуется как композиция двух более простых отображений

$$y = F(x) = F_{sem}(F_{sint}(x)). \qquad 3.2$$

Отображение F_{sint} называется синтаксическим отображением и связывает с каждой исходной программой некоторую структуру, которая служит аргументом семантического отображения F_{sem}. Описание КС–грамматиками синтаксиса исходного языка приводит к естественному использованию дерева грамматического разбора в качестве результата синтаксического отображения $F_{sint}(x)$. Тогда семантическое отображение обычно переводит дерево разбора исходного модуля в объектную программу.

Таким образом, будем рассматривать *два аспекта семантики языка программирования*:

• семантическую правильность программы, которая удовлетворяет контекстным условиям;

• правила семантического отображения исходной программы в результирующую; эти правила предназначены либо для описания процесса выполнения программы при ее интерпретации, либо для описания процесса ее перевода на машинный или машинно-ориентированный язык.

Для реализации семантики в языковом процессоре имеются специальные *семантические подпрограммы.*

Входная информация любого языкового процессора — это последовательность символов алфавита (обычно ASCII символы). В программе некоторые последовательности символов рассматриваются как единые объекты — лексические единицы языка (*лексемы*). Для выделения лексем из текста исходного модуля (ИМ) предназначен специальный блок языкового процессора - *лексический анализатор* или *сканер.* Резюмируя все вышесказанное, можно сделать вывод, что язык определяется тремя взаимосвязанными компонентами:

• лексикой,

• синтаксисом,

• семантикой.

Каждая составляющая языка реализуется в языковом процессоре соответствующим блоком или их совокупностью.

Лексика - это правила построения лексических единиц языка (лексем), т.е. слов, из которых строятся фразы языка. Для языков программирования обычно выделяют следующие типы лексем:

• ключевые слова,

• идентификаторы,

• константы,

• знаки операций,

• специальные знаки.

Как правило, эти группы лексем подразделяют на подгруппы.

Синтаксис — это правила построения цепочек языка из лексем, следовательно, синтаксис языка программирования описывается КС–грамматикой, терминальными символами которой являются лексемы.

3.2 Структура компилятора

Структура языкового процессора определяется структурой языка. На рис. 3.1 схематически представлена схема, соответствующая отображению (3.1). Система является однопроходной, т.к. процесс перевода осуществляется за один просмотр исходного модуля. Слева на рис. 3.1 изображена структура языка программирования, а справа от нее — блоки, реализующие обработку составляющих языка: лексика обрабатывается сканером, синтаксис — синтаксическим анализатором, контекстные условия — семантическими подпрограммами контроля, а правила семантического отображения исходного модуля в результирующий — семантическими подпрограммами перевода. Таким образом, главной структурной единицей компилятора является синтаксический анализатор, который по мере необходимости лексической и семантической обработки вызывает нужные подпрограммы.

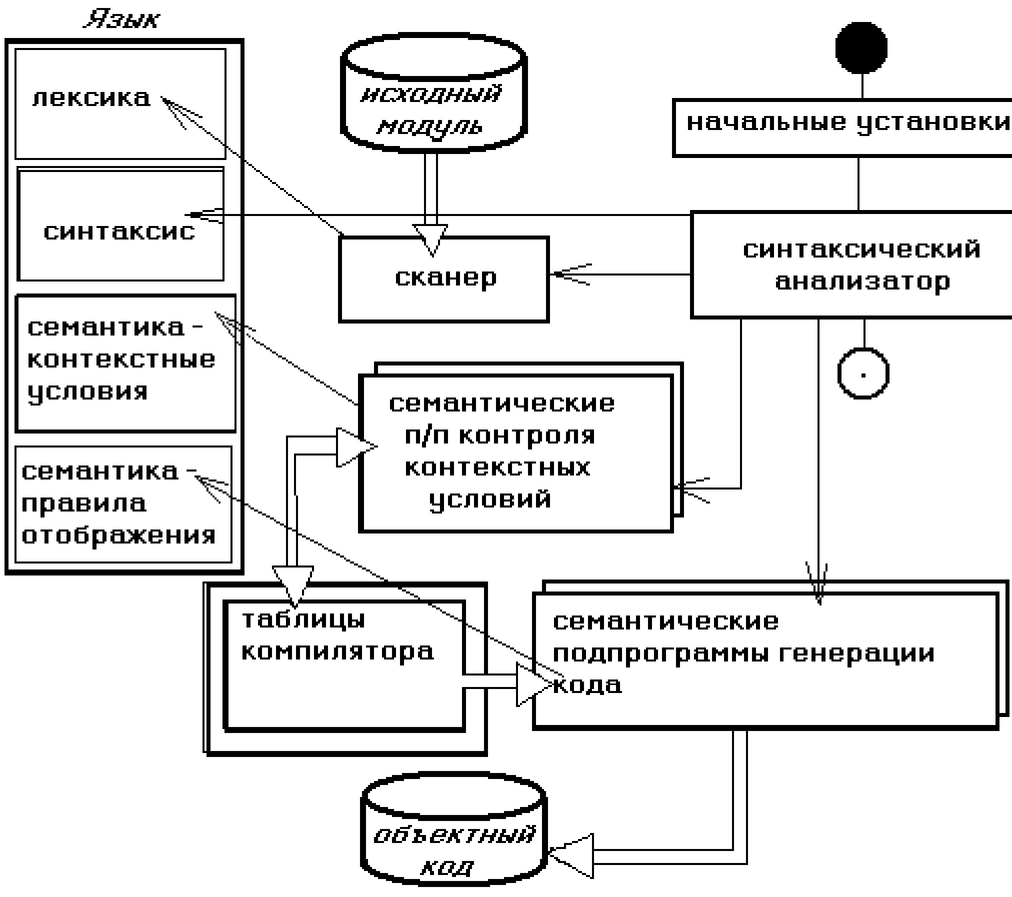


Рис. 3.1: Схема однопроходного языкового процессора

Под просмотром (или проходом) компилятора понимается процесс обработки всего, возможно, уже преобразованного, текста исходной программы. Одна или несколько фаз компиляции могут выполняться на одном просмотре. Двухпроходная схема отображения (3.2) основана на переводе исходного модуля в некоторую внутреннюю форму, а затем на переводе этой формы в результирующий модуль. Компиляторы, предназначенные для трансляции языков высокого уровня в объектный код, обычно оптимизируют этот код, тогда рассматриваем процесс перевода в виде сложного отображения

$$y = F(x) = F_1(F_2(...F_k(x)...)), \tag{3.3}$$

где функции F_i — это функции оптимизации и функции типа F_{sem} и F_{sint}. Получим классическую схему многопроходного компилятора, представленную на рис. 3.2.

Рис. 3.2: Схема многопроходного компилятора

Сравнивая однопроходную и многопроходную схемы трансляции, можно сделать вывод о следующих преимуществах многопроходной схемы по сравнению с однопроходной:

- возможность оптимизации;
- меньшая сложность программ анализа и синтеза.

Более простые алгоритмы программы анализа и синтеза объясняются следующими причинами:

• с блоком анализа не связываются элементы генерации,

• генератор рассматривает достаточно простые элементы полученного дерева разбора,

• имеется возможность использования одного блока генерации для разных языков программирования.

Рассмотрим подробнее назначение блоков компилятора, представленного на рис. 3.2. Исходной информацией для компилятора является цепочка символов (литер). *Лексический блок (сканер)* предназначен для того, чтобы разбивать цепочку символов на слова, из которых она состоит. Например, строка символов

```
if (alfa==1507) i++;
```

представляет собой последовательность слов "if", "(", "alfa", "==","1507", ")", "i", "++", ";". Исходный модуль с помощью программы сканера разбивается на отдельные *лексические единицы — лексемы*, которые поступают на *блок синтаксического анализа (или грамматического разбора) — синтаксический анализатор*. Анализатор строит дерево грамматического разбора, которое представлено в области данных компилятора в некоторой внутренней форме и используется теми блоками компилятора, которые работают на последующих шагах компидяции.

В процессе синтаксического анализа *выполняется семантический контроль с помощью набора семантических подпрограмм.* При этом используются *семантические таблицы компилятора*, в которых создается информация об объектах транслируемой программы. Таблицы также используются семантическими подпрограммами на фазе *генерации объектного кода.* Та часть компилятора, которая, строго говоря, не является необходимой, но позволяет получать более эффективные программы, называется блоками оптимизации генерируемой программы. Например, следует избегать повторной загрузки в регистр данных, удалить повторяющиеся фрагменты программы, вынести из цикла некоторые операции и др. Оптимизация может быть выполнена на различных уровнях представления оптимизируемой программы. Дерево разбора может быть преобразовано блоком *машино–независимой оптимизации.* После генерации оъектного кода может быть выполнена *машино–зависимая оптимизация.* Таким образом, оптимизационные преобразования возможны как на уровне дерева грамматического разбора, так и на уровне машинных команд.

Если компилятор не выполняет оптимизацию, то соответствующие блоки отсутствуют и используется двухпроходная схема. Если в таком компиляторе совместить анализ с синтезом, то при данном совмещении программ обработки получается уже рассмотренная ранее однопроходная схема. Разбиение на проходы может привести к дополнительным затратам памяти, т.к. каждое взаимодействие между проходами требует полной передачи данных от одного прохода к другому. Однако разбиение на проходы мотивируется весьма убедительными доводами, среди которых самыми существенными являются следующие.

1) *Логика языка.* Как правило, практически все языки программирования, в том числе даже языки Ассемблера, допускают так называемые ссылки вперед. Это значит, что в некоторый момент компилятору нужна информация из еще не рассмотренной части программы. Например, если описание идентификатора может появиться в тексте после его использования, то может случиться, что код нельзя сгенерировать до тех пор, пока не будет прочитан весь исходный модуль. В частности, при трансляции сложных программ транслятор *tasm.exe* потребует указания ключа двухпроходной компиляции.

2) *Простота реализации.* Разделение программы компилятора на отдельные логически законченные блоки приводит к ясной и простой структуре этой программы. Такую программу проще написать и отладить.

3) *Оптимизация кода.* Иногда объектный код получается более эффективным, если компилятору доступна информация обо всей программе в целом. Например, согласно некоторым методам оптимизации, нужно выделить все участки программы, где используются некоторые переменные и где могут изменяться их значения. Поэтому, прежде чем начинать оптимизацию, нужно просмотреть всю программу до конца.

4) *Экономия памяти.* Обычно многопроходные компиляторы занимают в памяти места меньше, чем компиляторы с одним проходом, т.к. программный код каждого прохода может вновь занимать память, занимаемую кодом предшествующего прохода.

Таким образом, реализация компилятора основана на проектировании и программировании пяти видов действий, выполняемых в процессе компиляции: лексический анализ, синтаксический анализ, семантическая обработка, оптимизация и генерация кода.

Вернемся к вопросу о количестве проходов транслятора. Обычно лексический анализ и синтаксический анализ выполняются на одном просмотре, т.е. синтаксический анализатор обращается к лексическому анализатору за очередной лексемой лишь по мере необходимости. иначе работают методы оптимизации кода, которые многократно просматривают программу. Передача информации между просмотрами происходит в терминах так называемых промежуточных языков. Структура промежуточного языка должна отображать синтаксическое дерево, и, может быть, какие-то внутренние таблицы компилятора. С одной стороны, увеличение количества проходов ведет к увеличению времени работы программы. С другой стороны, реализация транслятора с небольшим количеством проходов может привести к существенному усложнению алгоритмов трансляции. Например, $C\sharp$ позволяет использовать имя метода до того, как он был описан, следовательно, мы не можем выполнить семантический контроль вызова метода до тех пор, пока не будем знать имена и типы всех методов объектов. Таким образом, задачи сбора информации об обектах и проверки семантической корректности должны решаться на разных просмотрах.

3.3 Синтаксис языков программирования

Из теории формальных грамматик и языков известно, что один и тот же бесконечный язык может быть задан бесконечным числом различных грамматик. Среди этих грамматик могут быть и неоднозначные, т.е. такие, в которых можно построить несколько деревьев грамматического разбора для одной и той же цепочки языка. Какую грамматику для заданного языка следует выбрать — один из важнейших вопросов при построении компиляторов.

В описании синтаксиса языка программирования должно быть указано не только то, какие цепочки принадлежат языку, но также и то, как они должны выполняться. Поэтому в описании синтаксиса последовательности операторов должно учитываться, что такая последовательность выполняется слева направо, все описания данных также читаются слева направо. Правило определения выражений включает правила для определения операндов для каждой операции. В дереве грамматического разбора должно быть определено, какие подвыражения заданного выражения нужно вычислять на каждом шаге, каковы операнды каждой конкретной операции. В этом смысле дерево разбора представляет собой структуру транслируемой программы.

Транслятор выполняет синтаксический анализ исходного модуля по некоторому алгоритму грамматического разбора, поэтому структура КС–грамматики должна быть "подходящей" для метода разбора. В частности, восходящий разбор в укорачивающих грамматиках, как правило, сложнее по сравнению с разбором в неукорачивающих грамматиках, так как необходимо найти такой фрагмент входной цепочки, где можно вставить пустую цепочку ε. При восходящем разборе возникают сложности, если для различных нетерминалов имеются правила с одинаковыми правыми частями, т.к. необходимо решить дополнительную проблему выбора правильного нетерминала из тех, для которых существует указанное правило. При нисходящем разборе непреодолимые сложности возникают при анализе леворекурсивных КС–грамматик.

Поэтому, *структура КС–грамматики должна удовлетворять ограничениям, которые накладывает на структуру грамматики метод разбора.*

3.3.1 Программа

Аксиомой КС–грамматики, описывающей язык программирования, является тот нетерминал, которому соответствует обрабатываемый компилятором исходный модуль. Назовем этот нетерминал <программа>. Программа, написанная на разных языках программирования имеет различную структуру. Для иллюстрации сравним программы на языке Паскаль и языке C++.

На языке C++ программа представляет собой последовательность описаний. В качестве такого описания может выступать описание функции, описание данных или описание типа. Тогда соответствующие конструкции можно представить следующими правилами грамматики:

$$
\begin{array}{lll}
G_C: & <\text{программа}>\to & <\text{программа}><\text{описание}>|\varepsilon \\
 & <\text{описание}>\to & <\text{описание данных}>|<\text{описание типа}>| \\
 & & <\text{функция}> \\
 & <\text{описание данных}>\to & <\text{тип}><\text{список переменных}>; \\
 & <\text{тип}>\to & int|float|char... \\
 & <\text{список переменных}>\to & <\text{список переменных}>,<\text{данное}>|<\text{данное}> \\
 & ... &
\end{array}
$$

Программа, написанная на языке Паскаль, имеет более сложную организацию: сначала идут описания, а затем в операторных скобках $begin$ — end, которые заканчиваются знаком точки, расположено тело главной программы. Строгое описание языка регламентирует последовательность описаний констант, типов, данных, процедур и функций, причем именно в указанном порядке. Если следовать этому правилу, получается весьма сложная грамматика. Поэтому все разработчики компиляторов с языка Паскаль пошли по пути упрощения синтаксиса, а, следовательно, и сняли излишние ограничения на язык. Можно, например, использовать грамматику со следующими правилами:

$$
\begin{array}{lll}
G_P: & <\text{программа}>\to & <\text{заголовок}><\text{описания}> \\
 & & begin <\text{последовательность операторов}> end. \\
 & <\text{заголовок}>\to & program <\text{идентификатор}>;|\varepsilon \\
 & <\text{описания}>\to & <\text{описания}><\text{одно описание}> |\varepsilon \\
 & <\text{одно описание}>\to & <\text{константы}> \mid <\text{данные}> \mid <\text{описание типа}> \mid \\
 & & <\text{функция}> \mid <\text{процедура}> \\
 & <\text{данные}>\to & var <\text{список данных}> \\
 & <\text{список данных}>\to & <\text{список данных}><\text{переменные и тип}> \mid \\
 & & <\text{переменные и тип}> \\
 & <\text{переменные и тип}>\to & <\text{список переменных}>:<\text{тип}>; \\
 & <\text{тип}>\to & integer|real|boolean|char... \\
 & <\text{список переменных}>\to & <\text{список переменных}>,<\text{данное}> \mid <\text{данное}> \\
 & ... &
\end{array}
$$

Использование понятий в угловых скобках в качестве нетерминалов позволяет наглядно описывать структуру программы, но, к сожалению, весьма громоздко и может вызвать ошибки при программировании синтаксических анализаторов. Поэтому в дальнейшем в качестве нетерминалов будем использовать большие латинские буквы, поясняя, если требуется, их назначение. Например, указанные выше правила грамматики в такой записи примут вид:

$$
\begin{array}{ll}
G_P: & P \to ZWbegin\ H\ end. \\
 & Z \to program\ I;|\varepsilon \\
 & W \to WY|\varepsilon \\
 & Y \to C|D|T|F|P \\
 & D \to var\ K \\
 & K \to KU|U \\
 & U \to L : A; \\
 & A \to integer|real|boolean|char... \\
 & L \to L, X|X \\
 & ...
\end{array}
$$

3.3.2 Выражения

Пожалуй, наибольшую сложность в языках программирования имеют выражения. И дело здесь не только в скобочной структуре выражения. Для того, чтобы продемонстрировать основные сложности, рассмотрим сначала простую КС–грамматику, порождающую все правильные выражения, содержащие идентификаторы простых переменных, константы, знаки бинарных арифметических операций "+", "-", "*", "/", круглые скобки. Такая простейшая КС–грамматика может иметь вид:

$$G : S \to S + S|S - S|S * S|S/S|a|c|(S),$$

где символы a и c соответственно обозначают идентификаторы и константы. Очевидно, что предложенная КС–грамматика порождает те и только те цепочки, которые являются правильными выражениями, удовлетворяющими поставленным ограничениям на операнды и знаки операций. Рассмотрим особенности построенной КС–грамматики.

Как известно, в КС–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. Определить понятие эквивалентности двух выводов для заданной КС–грамматики можно на основе графического представления вывода, называемого деревом вывода или деревом грамматического разбора. Для любой цепочки дерево вывода отражает ее структуру.

Сначала отметим, что хотелось бы иметь грамматику, которая в той или иной мере отражала бы смысл каждой цепочки. Конечно, синтаксическая конструкция сама по себе в общем случае не может приписывать смыл соответствующим цепочкам языка. Такое приписывание смысла может появиться только в процессе интерпретации или компиляции построенной синтаксической конструкции. Семантика языка определяется в зависимости от контекста и должна реализовываться семантическими программами транслятора. Грамматика может только описывать синтаксические конструкции языка и не имеет средств для описания семантики. Но у грамматики есть и другая важная функция: *построенная грамматика должна иметь такую структуру, чтобы максимально упростить семантические программы транслятора.*

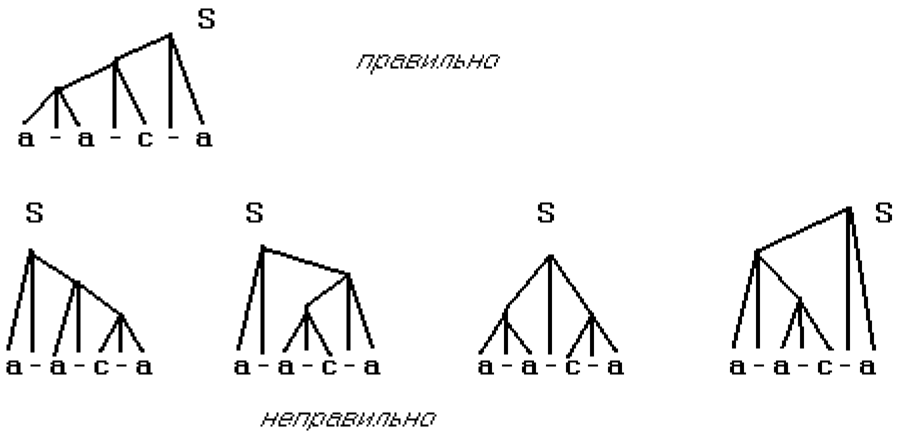


Рис. 3.3: Нарушение порядка выполнения операций одного приоритета в КС–грамматике $G : S \to S + S|S - S|S * S|S/S|a|c|(S)$

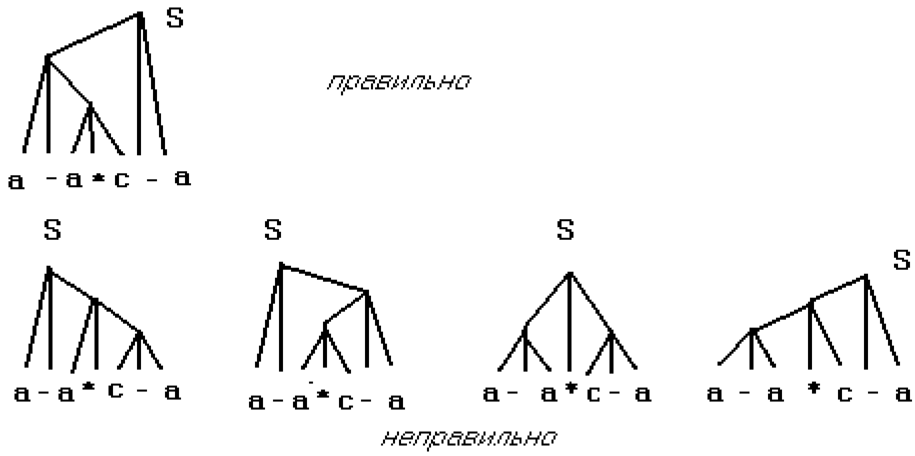


Рис. 3.4: Нарушение приоритетов операций в КС–грамматике $G : S \to S+S|S-S|S*S|S/S|a|c|(S)$

Посмотрим теперь критически на построенную нами грамматику арифметических выражений. Разумно предполагать, что дерево грамматического разбора выражения отображает порядок операций при вычислении этого выражения. Тогда примеры на рис. 3.3 и рис. 3.4 показывают абсолютную непригодность этой грамматики: операции одного приоритета могут выполняться в нарушение порядка " слева — направо", операции могут выполняться не в соответствии с их приоритетами. Ситуация не улучшится, даже если ввести ограничения на способ построения дерева разбора — например, всегда строить левое или правое дерево разбора. Такое дерево будет единственным, но семантически неправильным.

Какую же грамматику выбрать для описания выражений? Мы теперь знаем, что такой выбор зависит от подразумеваемой семантики цепочек языка. Поэтому в зависимости от языка программирования, синтаксис которого мы будем описывать, грамматики могут быть разными. Эти различия должны определяться следующими особенностями языка программирования:

• правилами приоритетов операций;

• правилами выполнения операций одного приоритета;

• типами операндов выражений;

• правилами записи унарных и бинарных операций, а также особенностями выполнения этих операций.

Все эти особенности языка программирования будут отражены в КС–грамматике, которую построим по следующему алгоритму. Итак, рассмотрим следующий *алгоритм построения КС–грамматики для описания выражений языков программирования.*

1. Упорядочим все операции в порядке возрастания их приоритета. Пусть минимальный приоритет равен 1.

2. Каждому уровню приоритета i поставим в соответствие нетерминал A_i. Отметим, что понятию "выражение" соответствует нетерминал при младшем уровне приоритетов A_1.

3. Добавим еще один нетерминал A_{n+1}, который будет соответствовать понятию "элементарное выражение".

4. Для каждого нетерминала A_i $(1 \le i \le n)$ и всех операций данного приоритета i записывается правило:

— для бинарной операции (обозначим операцию здесь и далее знаком $\oplus$), выполняющейся слева направо, правило вида

$$A_i \to A_i \oplus A_{i+1},$$

— для бинарной операции, выполняющейся справа налево

$$A_i \to A_{i+1} \oplus A_i,$$

— для унарной префиксной операции однократного использования

$$A_i \to \oplus A_{i+1},$$

— для унарной префиксной операции многократного использования

$$A_i \to \oplus A_i,$$

— для унарной постфиксной операции однократного использования

$$A_i \to A_{i+1}\oplus,$$

— для унарной постфиксной операции многократного использования

$$A_i \to A_i \oplus .$$

5. Для каждого нетерминала, кроме последнего A_{n+1}, соответствующего понятию "*элементарное выражение*", добавляется правило

$$A_i \to A_{i+1}.$$

6. Для нетерминала A_{n+1}, соответствующего понятию "элементарное выражение", записываются правила двух типов:

— для всех элементарных операндов — идентификаторов, констант, вызовов функций и т.п. — правила

$$A_{n+1} \to < \textit{идентификатор} > | < \textit{константа} > | < \textit{функция} > |...,$$

— если в выражении допускается скобочная структура, то правило

$$A_{n+1} \to (A_1).$$

Этот метод работает безотказно в любом случае, Следует, однако, сделать замечание, касающееся унарных операций "+" и "-", которые совпадают по написанию с бинарными операциями. Как правило, не разрешается писать такие операции подряд, например, в языке Паскаль не допускается запись типа

```
a := b++c--(d+-3);
```

Для того, чтобы указать, что такие операции можно использовать только в начале выражения (в том числе и после открывающейся круглой скобки), эти операции поднимают вверх в таблице приоритетов до начального понятия <выражение>, и считают, что эти знаки могут стоять только в начале конструкции <*выражение*> или после открывающейся скобки.

Например, если можно использовать бинарные знаки операций $+$, $-$, $*$, $/$, то они упорядочиваются по приоритетам следующим образом:

1) операции минимального приоритета $+$, $-$,

2) операции максимального приоритета $*$, $/$.

Этим уровням приоритетов соответствуют два уровня нетерминалов A_1 и A_2, а также дополнительный нетерминал A_3, соотвествующий понятию элементарное выражение. Все эти бинарные операции выполняются слева направо, поэтому получаем грамматику:

$$\begin{aligned} G: \quad & A_1 \to A_1 + A_2 | A_1 - A_2 | A_2 \\ & A_2 \to A_2 * A_3 | A_2 / A_3 | A_3 \\ & A_3 \to a | c | (A_1), \end{aligned}$$

где для наглядности символами a и c соответственно обозначены идентификатор и константа. Чтобы работать с неиндексированными нетерминалами, переобозначим их символами V, A, T, получим грамматику:

$$\begin{aligned} G: \quad & V \to V + A | V - A | A \\ & A \to A * T | A / T | T \\ & T \to a | c | (V), \end{aligned}$$

Если к множеству операций добавить унарные знаки операций $+$ и $-$, то они будут поставлены на первый уровень. Тогда грамматика примет вид:

$$\begin{aligned} G: \quad & V \to V + A | V - A | A | -A | +A \\ & A \to A * T | A/T | T \\ & T \to a | c | (V), \end{aligned}$$

Сделаем еще некоторые замечания относительно знаковых и беззнаковых констант. Как правило, все компиляторы рассматривают беззнаковые константы, а знак "-" перед константой рассматривается как унарная операция изменения знака. Причина этого — невозможность определить разницу между знаком константы и знаком операции вне контекста. Например, в выражении

$$-4 - (+5 + 6) - (-7 - 8)$$

имеется знаковые константы "-4", "+5", "-7", а также беззнаковые константы "6" и "8". Выделение сканером констант со знаком привело бы к необходимости анализа контекста, что усложняет работу сканера.

3.3.3 Структурные операторы

Рассмотрим упрощенную структуру оператора if языка C++, внутри которого могут быть только такие же операторы if, составной оператор и простейший оператор присваивания. Пусть в выражении допускается использование унарных и бинарных операций "+" и "-", бинарных "*" и "/". Элементарными операндами являются константы и идентификаторы. Обозначим символами a и c идентификатор и константу соответственно. Тогда грамматика имеет вид

$$\begin{aligned} G: \quad & S \to if(V)O \mid if(V)O \; else \; O \\ & O \to S \mid a = V; \mid \{H\} \\ & H \to HO \mid \varepsilon \\ & V \to +A \mid -A \mid V + A \mid V - A \mid A \\ & A \to A * T \mid A/T \mid T \\ & T \to a \mid c \mid (V) \end{aligned}$$

Сразу отметим, что построенная конструкция оператора if является неоднозначной (см. примеры грамматического разбора на рис. 3.5), однако решение проблемы однозначного разбора этого оператора мы переложим на синтаксический анализатор и рассмотрим в дальнейшем. Пока же только отметим, что однозначный разбор будет выполняться по правилу "каждый $else$ соответствует предшествующему $then$". В языке Паскаль имеются оба ключевых слова $else$ и $then$, в языке C++ и ему подобных языках вместо слова $then$ используется правая круглая скобка, в которую заключается выражение. Сопоставить слово $else$ при грамматическом разборе можно и со словом if, все зависит, как мы увидим в дальнейшем, от метода разбора.

Остальные структурные операторы не представляют сложностей в записи правил и не содержат внутренних причин для неоднозначного представления. Например, достаточно упрощенную структуру операторов цикла в языке C++ можно представить совокупностью правил грамматики

$$\begin{aligned} G: S \to \quad & do\{H\} \; while(V); | while \; (V) \; O | \\ & for \; (a = V; \; V; \; a = V) \; O, \end{aligned}$$

где H — последовательность операторов, O — один оператор.

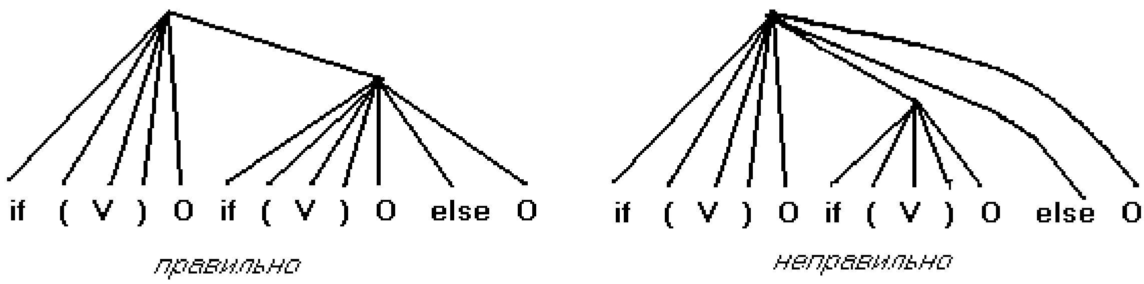


Рис. 3.5: Неоднозначный разбор условного оператора

3.3.4 Составной оператор

Рассмотрим подробнее структуру составного оператора. В любом языке программирования, допускающем использование составных операторов, используются специальные операторные скобки для выделения всех операторов, которые находятся внутри составного оператора. Это, например, ключевые слова *begin* и *end* в языке Паскаль, знаки фигурных скобок в языках C++, Перл или Java. При этом в разных языках существенно различается перечисление операторов внутри составного оператора. Так, например, в языке Паскаль оператор отделяется от оператора знаком "точка с запятой":

$$H \to H; O|O,$$

где, как и ранее, H — последовательность операторов, а O — один оператор. Пустой оператор в языке Паскаль — это просто пустая цепочка. Таким образом, правила описания оператора языка Паскаль имеют вид

$$O \to \varepsilon|\ a := V \ |\ begin\ H\ end\ |...$$

Если перед ключевым словом *end* стоит знак "точка с запятой это означает, что последним оператором перед *end* является пустой оператор.

В языке программирования C++ и многих других языках знак "точка с запятой" является не разделителем между операторами, а ограничителем простейшего оператора, например, оператора присваивания или вызова функции. В этом случае последовательность операторов состоит из операторов, между которыми нет специального знака:

$$\begin{array}{ll} H & \to HO|\varepsilon \\ O & \to;\ |\ a := V;\ |\ \{\ H\ \}\ |\ ... \end{array}$$

Очевидно, что пустой оператор в этом случае содержит единственный символ — знак "точка с запятой".

3.4 Контекстные условия языков программирования

Как уже отмечалось выше, требования, связанные с корректностью использования объектов в программе, находятся за пределами возможностей КС–языков. Считается, что эти требования относятся к семантике языка, и для их контроля в процессе синтаксического анализа вызываются семантические подпрограммы. Рассмотрим контекстные условия, контролируемые с помощью семантических подпрограмм.

Большинство контекстных условий связано с двумя важнейшими особенностями языков программирования. Первой такой особенностью является использование различных типов данных. В большинстве языков по изображению идентификатора нельзя определить тип значения, которое данный идентификатор именует в программе. Поэтому информация о типах идентификаторов указывается в программе явно с помощью описаний. Второй особенностью, приводящей к необходимости проверки контекстных условий, является понятие так называемой области действия (или области видимости). Идентификаторы могут использоваться только внутри своей области действия.

В соответствии с той ролью, которую играют контекстные условия в описании семантики языков программирования, они могут быть разбиты на группы. Обычно выделяют следующие типы семантических (контекстных) условий.

1) *Каждый объект должен быть описан.* Языки разделяются на языки с умолчанием (например, Бейсик) и языки без умолчания (например, C++ и Паскаль). В языке без умолчания каждый объект должен быть описан ровно один раз, а в языке с умолчанием объект можно описать не более одного раза. Например, нельзя написать

```
int * vector, i, i, vector[100];
```

2) *Область использования объекта должна быть согласована с областью его действия.* Понятие области видимости объекта — одно оз важнейших свойств языка программирования. Это означает, что, например, нельзя описать переменную внутри некоторого блока, а потом использовать эту переменную вне данного блока. Например, пусть в программе имеются следующие описания данных:

```
int i;
void main (void)
{
int j;
i=1; j=2; // присваивание значений глобальной переменной i и локальной j
   {
      int i,k;
      i=3; k=4;              // локальные в новом блоке переменные i и k
   }
   {
      k=3;            // ошибка: недоступная в новом блоке переменная k
   }
}
```

В этой программе описаны четыре переменные. Во втором внутреннем блоке использование переменной k недопустимо, т.к. ее область видимости — только первый внутренний блок.

3) *Типы формальных и фактических параметров процедур и функций и их количество должны совпадать.* Хотя иногда вводят различие между передачей параметров по наименованию (по ссылке или адресу) и по значению, в общем случае это различие просто означает разные типы парамеров. Например, в функции

```
void funSum (int n, int * data);
```

параметр n записывается в стек как целое число, а параметр $data$ представляет собой адрес, который тоже записывается в стек. Поэтому говорят, что в языке C++ все параметры передаются только по значению. Требование совпадения типов параметров или приводимости типа формального параметра к типу фактического параметра касается практически всех языков программирования. Более того, чаще всего встречается именно точное совпадение типов, а приведение если и допускается, то не для произвольных типов данных. Например, в языке Паскаль нельзя написать

```
var a: real;
procedure Proc(Param: integer);
   begin
      ...
   end;
begin
   Proc(a);  {  ошибочный параметр: real не приводится к integer  }
end.
```

Каждый язык программирования имеет свою собственную систему приведения типов, в соответствии с которой выполняются приведения в разном контексте, в том числе и для фактических параметров процедур и функций. В частности, язык C++ допускает автоматическое приведение любого типа к типу *void*, например, допустима запись

```
double m[MAX_N];
int f1(void * a)   {
   if ( a == NULL ) return 1;
   return 0;
}
int main(void)   {
    ...
   if (f1(m)) printf("Адрес m равен нулю!\n");
}
```

Иногда язык программирования допускает явное указание приведения, как, например, это реализовано в языке C++. Даже если в язык программирования встроены достаточно мощные правила приведения типов, автоматическое нетривиальное приведение типов фактических параметров не разрешается. Например, функция сортировки

```
void qsort(  void *base,
             size_t num,
             size_t width,
             int (__cdecl *compare )(const void *, const void *)
          ) ;
```

требует создания функции-компаратора с двумя параметрами типа *const void **, поэтому в коде этой функции необходимо указать явное приведение типов:

```
int sort_function( const void *a, const void *b) {
   // int na=*a; int nb=*b;  --- ошибка приведения типов!
   int na=*((int *)a); int nb=*((int *)b);  // правильно
   if ( fabs(data[na] - data[nb]) < eps)  return 0;
```

```
   else
   if  ( data[na] < data[nb]) return 1;
   else return -1;
}
```

4) *Использование объектов должно быть согласовано как с типом оператора, в котором этот объект встречается, так и с типом операции, которая выполняется над объектом.* Например, ошибочным является фрагмент программы на языке С++

```
FILE * in, *out;
int *  a;
out=in%5;   // неверное использование указателя с операцией %
a+=5;       // семантически допустимая операция
```

5) *Типы объектов, связанных некоторой операцией, должны быть согласованы.* Например, функция сортировки *qsort*(...) в качестве первого параметра требует адрес сортируемого массива как *void*∗, поэтому приходится указывать явное преобразование типа *int*∗ к типу *void*∗:

```
int  vector[1000];
qsort( (void *)vector, (size_t)vector, sizeof( int ), compareFunction );
```

Большинство типов данных имеет различное представление на уровне машинного кода. Приведение типов выполняется компилятором для того, чтобы в процессе компиляции в объектный код можно было вставить специальные операции приведения типов. Это или одна команда (например, для преобразования *short int* в *long int*) или вызов подпрограммы преобразования (например, для преобразования *short int* в *double*). В некоторых случаях такое преобразование не выполняется, но над одной и той же последовательностью битов выполняются разные машинные команды в зависимости от типов операндов. Например, при выполнении следующей программы будут выведены два различных значения $a = -2.000000$ и $b = 4294967296.000000$, хотя, казалось бы, переменным прсваиваются одни и те же значения:

```
int  i,j;
unsigned int ui,uj;
float a,b;
int main(void)
{
  i =-2;   ui = i;
  j = 5;   uj = j;
  printf("i=%d    ui=%d\n",i,ui); // представление определяется форматом %d,
                                  // поэтому вудут выведены одинаковые числа
  a = i;   b  = ui;               // выполняется приведение типов
  printf("a=%f    b=%f\n",a,b);   // выводятся разные значения
}
```

Информация для контроля контекстных условий должна храниться в таблицах транслятора. Структуру этой информации мы рассмотрим в дальнейшем.

3.5 Типы синтаксических анализаторов

Рассмотрим методы программирования синтаксического анализатора синтаксически–ориентированного транслятора. Эти методы делятся на два класса:

• общие методы анализа,

• специальные методы.

Общие методы работают на всех КС–грамматиках, специальные — только на некоторых типах грамматик, специально выделенных для каждого метода. В силу известной теоремы об эквивалентности КС–языков и недетерминированных автоматов с магазинной памятью, универсальные методы анализа представляют собой детерминированные модели недетерминированных магазинных методов. Поэтому общие методы — это нисходящие и восходящие разборы с возвратами, реализованные в виде рекурсивных процедур. Общие методы синтаксического анализа могут быть применены для любого КС–языка, однако, при построении трансляторов языков программирования они не применяются по двум причинам:

• они требуют огромных затрат времени в силу недетерминированного алгоритма (как известно, реализация программы, соответствующей недетерминированным методам работы, приводит к алгоритмам полного перебора, которые обладают экспоненциальной временной сложностью);

• поскольку методы возвратны, на каждом шаге возврата необходимо реализовать отмену выполненных действий, связанных с семантическим вычислением, например, удаление некоторой информации из таблиц транслятора или отказ от уже выданного сообщения об ошибке. Это весьма трудоемко.

Для трансляции алгоритмических языков требуются эффективные детерминированные алгоритмы синтаксического анализа. Очевидно, что такие методы могут быть использованы только при некоторых ограничениях на КС–грамматики. Эти ограничения для каждого метода определяют некоторый собственный специальным образом определенный класс КС–грамматик, поэтому такие методы называются специальными методами анализа.

С точки зрения функционирования специальные методы можно подразделить на два типа:

• выполняющие восходящий разбор (методы предшествования, $LR(k)$ и др.);

• выполняющие нисходящий разбор (метод рекурсивного спуска, $LL(K)$ и др.).

3.6 Контрольные вопросы к разделу 1

1. Перечислите контекстные условия языков программирования.
2. Перечислите типы трансляторов. Что положено в основу их классификации?
3. Зачем используется сканер?
4. В чем заключается задача, решаемая на фазе лексического анализа?
5. Как строится КС–грамматика, описывающая выражения, сконструированные на основе приоритетов операций?
6. Как определяется синтаксис составного оператора в различных языках программирования?
7. Чем определяется семантика языка программирования?
8. Дайте определение языкового процессора.

9. С помощью каких составных элементов компилятора проверяются контекстные условия?

10. Перечислите типы синтаксических анализаторов.

11. Почему при трансляции языков программирования высокого уровня работа выполняется по принципу "несколько в несколько"?

12. Какой языковый процессор называется генератором?

13. Какая программа называется компилятором? Чем компилятор отличается от макрогенератора?

14. Постройте структурную схему однопроходного компилятора.

15. Постройте структурную схему многопроходного компилятора. Чем многопроходной компилятор отличается от однопроходного компилятора?

16. Что называется интерпретатором? Приведите примеры известных Вам интерпретаторов.

17. Запишите отображения, которые реализуются в процессе однопроходной и многопроходной трансляции.

18. Какую роль выполняют семантические подпрограммы компилятора?

19. Что называется прямым компилятором? Приведите пример алгоритмов прямой компиляции.

20. В чем состоит задача синтаксического анализа и какой блок компилятора решает эту задачу?

3.7 Тесты для самоконтроля к разделу 1

1. Чем однопроходной компилятор отличается от многопроходного?

Варианты ответов:

а) многопроходной компилятор выполняет синтаксический анализ исходного модуля в процессе многократного сканирования этого модуля; однопроходной компилятор сканирует текст только один раз;

б) однопроходной компилятор выполняет синтаксический анализ, оптимизацию кода и генерацию объектного кода для каждого единичного оператора в отдельности, а многопроходной компилятор делает это для многих операторов сразу;

в) многопроходной компилятор в процессе синтаксического анализа генерирует некоторый внутренний код, который затем многократно просматривает в процессе оптимизации и генерации объектного кода;

г) многопроходной компилятор работает по принципу ”много в несколько”, а однопроходные — по принципу ”один в несколько”

д) многопроходной компилятор работает по принципу ”много в очень много”, а однопроходные — по принципу ”один в один”

Правильный ответ: в.

2. В языке программирования допускаются выражения со скобочной структурой, которые содержат операнды и операции. Операции — это бинарные и унарные знаки ”+”, ”-”, унарные префиксные и постфиксные ”++” и ”–”. Все унарные операции однократного применения. Операции выполняются слева направо. Операнды — это идентификаторы и вызовы функций без параметров (знак идентификатора — символ a). Напишите КС–грамматику, описывающую такие выражения.

Варианты ответов:

$$
\begin{array}{lll}
\text{а)} & G: & S \to A|S+S|S-S \\
 & & A \to a|(A)|a()|-A|--A|A--|A++|+A|++A \\
\text{б)} & G: & S \to -S|+S|S-|S+|S+A|S-A \\
 & & A \to a|(S)|a() \\
\text{в)} & G: & S \to -A|+A|A--|A++|S+A|S-A \\
 & & A \to a|(S)|a() \\
\text{г)} & G: & S \to -S|+S|A--|A++|S+A|S-A \\
 & & A \to a|(S)|a() \\
\text{д)} & G: & S \to -A|+A|A--|A++|--A|++A|S+A|S-A|A \\
 & & A \to a|(S)|a() \\
\text{е)} & G: & S \to -A|+A|A--|A++|S+A|S-A|A \\
 & & A \to a|(S)|a()
\end{array}
$$

Правильный ответ: д.

3. Зачем используются семантические подпрограммы?

Варианты ответов:

а) семантические подпрограммы нужны для оптимизации кода на различных уровнях представления транслируемой программы;

б) использование семантических подпрограмм позволяет существенно сократить объем КС–грамматики, с помощью которой описывается синтаксис языка программирования, т.к. часть синтаксических условий выносится на уровень контроля семантическими подпрограммами;

в) структура КС–грамматики должна удовлетворять ограничениям, которые накладывает на структуру грамматики метод разбора, а при использовании семантических подпрограмм можно существенно уменьшить указанные ограничения;

г) семантические подпрограммы позволяют правильно построить дерево грамматического разбора, указывая позиции в этом дереве, предназначенные для операндов требуемого типа;

д) в процессе синтаксического анализа выполняется семантический контроль контекстных условий с помощью набора семантических подпрограмм.

Правильный ответ: д.

4. Какие из следующих утверждений истинны?

1) Компиляторы, предназначенные для трансляции языков высокого уровня в объектный код, обычно оптимизируют этот код, тогда рассматриваем процесс перевода в виде сложного отображения

$$y = F(x) = F_1(F_2(...F_k(x)...)),$$

2) схема перевода "несколько в очень много" основана на отображении

$$y = F(x) = F_1(F_2(...F_k(x)...)),$$

3) однопроходная схема компилятора характеризуется более простыми алгоритмами программы анализа и синтеза по сравнению с многопроходной;

4) дерево грамматического разбора строится компилятором в том случае, если требуется оптимизация кода;

5) главной структурной единицей компилятора является синтаксический анализатор, который по мере необходимости лексической и семантической обработки вызывает нужные подпрограммы.

Варианты ответов:

а) все утверждения ложны;

б) 2 и 5;

в) 3 и 4;

г) 1, 3 и 4;

д) 1 и 5;

е) 2, 3 и 4;

ж) 2, 3 и 5;

з) все утверждения истинны.

Правильный ответ: д.

5. Поясните назначение сканера.

Варианты ответов:

а) сканер предназначен для игнорирования комментариев и других незначащих символов в тексте исходного модуля;

б) сканер читает файл с текстом исходного модуля;

в) сканер помогает синтаксическому анализатору правильно указать место в программе, где допущена ошибка;

г) сканер выполняет синтаксический анализ исходного модуля;

д) исходный модуль с помощью программы сканера разбивается на отдельные лексемы, которые поступают на блок синтаксического анализа.

Правильный ответ: д.

3.8 Упражнения к разделу

3.8.1 Задание

Цель данного задания – построить КС–грамматику, описывающую заданный язык программирования. Затем необходимо выделить лексический и синтаксический уровень грамматики. Работу над заданием следует организовать, последовательно выполняя следующие операции.

1. Для языка программирования, указанного в Вашем задании, определить синтаксическую конструкцию, соответствующую главной программе. Построить правила КС-грамматики, определяющие программу в целом и место в ней описаний и операторов,

2. Построить правила КС-грамматики, определяющие синтаксис отдельных операторов и описаний.

3. Проанализировать список операций (арифметических, логических, сравнения и т.п.), которые должны использоваться в языке программирования в соответствии с Вашим заданием. Построить таблицу приоритетов операций. Отметить бинарные и унарные операции.

4. Построить правила КС-грамматики для выражений. При определении понятия элементарного выражения учесть все типы констант и простых операндов (из числа простых переменных, элементов массивов, вызовов функций, полей структур и т.п.).

5. Построить правила, определяющие синтаксис констант, которые могут быть использованы в языке программирования Вашего задания.

6. Определить лексический и синтаксический уровень Вашей грамматики. Для этого в каждом правиле КС-грамматики выделить все лексические единицы. Затем отметить все правила, определяющие выделенные лексемы и правила более низкого уровня.

3.8.2 Пример выполнения задания

Рассмотрим в качестве примера построение грамматики для некоторого простейшего варианта языка Java-script. Введем правила описания

1) программы в целом;
2) типов данных;
3) используемых в выражниях операций;
4) операторов, которые могут встречаться в выполняющейся части программы;
5) операндов, используемых в выражениях;
6) констант.

Итак, пусть программа — это множество функций и описаний данных, которые заключены в специальные скобки комментариев HTML–документа

$$<! - - \text{ и } - - >$$

и включены еще в одни внешние скобки

$$< SCRIPT\ language =''\ JavaScript'' > \text{ и } < /SCRIPT >$$

Синтаксис языка Java–Script очень похож на синтаксис языка C++, но структура программы существенно проще. Данные описываются без указания типа с помощью оператора описания var. Описание состоит из ключевого слова var, за которым стоят идентификаторы переменных с возможной инициализацией. Язык Java–Script — объектно–ориентированный язык, в качестве объектов выступают как сам HTML–документ, так и объекты этого документа. Сейчас нас будет интересовать не семантика этих объектов, а только синтаксис обращения к ним. Синтаксически такое именование представляется последовательностью идентификаторов, разделенных знаками точки (как, например, структуры в языке C++). Таким образом, операндами выражений являются простые переменные, элементы структур, константы и вызовы функций.

Тип переменная получает только после того, как ей будет присвоено значение. Таким типами данных могут быть целые, вещественные и строковые данные. Соответственно, в качестве констант могут выступать константы целые в десятичной системе счисления (10 $c\backslash c$), вещественные, а также строковые константы, представляющие собой произвольную последовательность символов, заключенную в двойные кавычки (как в языке C++).

Для простоты в качестве операций будем использовать только арифметические операции и операции сравнения.

В качестве операторов рассмотрим также только некоторое ограниченное подмножество операторов языка Java–Script: присваивания, if, for, пустой оператор. Синтаксис этих операторов полностью совпадает с синтаксисом аналогичных операторов языка C++.

Описание функций похоже на описание функций в языке C++, за тем исключением, что в соответствии с соглашением о типах тип формальных параметров не описывается. Таким образом, список формальных параметров представляет собой только список идентификаторов, разделенных запятыми.

Перейдем теперь к описанию с помощью КС–грамматики синтаксиса этого языка. Поскольку наш язык программирования содержит знаки $<$ и $>$ в качестве терминальных символов, то для повышения наглядности КС–грамматики будем обозначать все терминалы выделенными символами.

Итак, программа — это последовательность описаний, заключенная в двойные специальные скобки:

$$
\begin{array}{lll}
G_J: & <\text{программа}>\to & <\textbf{SCRIPT language} ='' \textbf{JavaScript}''> \\
 & & <!-- \\
 & & <\text{описания}> \\
 & & --> \\
 & & <\textbf{/SCRIPT}> \\
 & <\text{описания}>\to & <\text{описания}>\ <\text{одно описание}>|\varepsilon \\
 & <\text{одно описание}>\to & <\text{данные}>|<\text{функция}>
\end{array}
$$

Описание данных начинается ключевым словом var, за которым следует список переменных:

$$
\begin{array}{ll}
<\text{данные}>\to & \textbf{var} <\text{список}>; \\
<\text{список}>\to & <\text{список}>, <\text{переменная}>|<\text{переменная}> \\
<\text{переменная}>\to & <\text{идентификатор}>|<\text{идентификатор}>=<\text{выражение}>
\end{array}
$$

Функция построена из заголовка и тела функции, в качестве которого используется составной оператор. Составной оператор в большинстве языков программирования — это последовательность операторов и описаний переменных, заключенная в операторные скобки. В качестве операторных скобок в языке Java–Script используются фигурные скобки.

$$
\begin{array}{ll}
<\text{функция}>\rightarrow & \textbf{function}(<\text{список}>)<\text{составной оператор}> \\
<\text{составной оператор}>\rightarrow & \{<\text{операторы и описания}>\} \\
<\text{операторы и описания}>\rightarrow & <\text{операторы и описания}><\text{данные}>| \\
 & <\text{операторы и описания}><\text{оператор}>|\varepsilon \\
<\text{оператор}>\rightarrow & <\text{присваивание}>;|<\text{составной оператор}> \\
 & <\text{вызов функции}>|<for>|<if>|\,; \\
<for>\rightarrow & \textbf{for}(<\text{присваивание}>;<\text{выражение}>; \\
 & <\text{присваивание}>)<\text{оператор}> \\
<if>\rightarrow & \textbf{if}(<\text{выражение}>)<\text{оператор}>| \\
 & \textbf{if}(<\text{выражение}>)<\text{оператор}> \\
 & \textbf{else}<\text{оператор}>
\end{array}
$$

В операторе присваивания значение можно присвоить как простой переменной, так и элементу структуры. Поэтому введем понятие $<$имя$>$ в качестве обозначения последовательности идентификаторов, разделенных точками. Тогда определение оператора присваивания имеет вид:

$$
\begin{array}{ll}
<\text{присваивание}>\rightarrow & <\text{имя}>=<\text{выражение}> \\
<\text{имя}>\rightarrow & <\text{имя}>.<\text{идентификатор}>|<\text{идентификатор}>
\end{array}
$$

Конструкция выражения определяется операциями и операндами, из которых построено это выражение. Сначала определим типы элементарных операндов. Пусть это будут

а) имена,

б) все константы — целые, вещественные, строковые,

в) вызовы функций. Функции могут иметь фактические параметры — произвольные выражения.

Теперь упорядочим по возрастанию приоритетов операции, которые можно использовать в выражении. При этом необходимо решить, на какой уровень приоритетов мы поставим унарные знаки $+$ и $-$. Пусть эти знаки можно использовать только в начале выражения и после круглых скобок, тогда их необходимо поместить на первый уровень. Получим следующую таблицу приоритетов:

1) минимальный приоритет у всех знаков сравнения: $<$, $<=$, $>$, $>=$, $==$, $!=$; здесь же находятся унарные $+$ и $-$;

2) следующий уровень приоритетов у знаков бинарных аддитивных операций $+$ и $-$;

3) максимальный уровень имеют мультипликативные операции $*$, $/$, $\%$.

Поставим в соответствие этим трем уровням приоритетов нетерминалы

$<$выражение$>$

$<$слагаемое$>$

$<$множитель$>$.

Сразу следует отметить тот важный факт, что именно нетерминал первого уровня приоритетов и должен получить наименование ”выражение”. Добавим нетерминал

<элементарное выражение>, получим правила грамматики для выражений:

$$
\begin{array}{ll}
<\text{ выражение }>\to & <\text{ выражение }> \; > \; <\text{ слагаемое }> | \\
 & <\text{ выражение }> \; >= \; <\text{ слагаемое }> | \\
 & <\text{ выражение }> \; < \; <\text{ слагаемое }> | \\
 & <\text{ выражение }> \; <= \; <\text{ слагаемое }> | \\
 & <\text{ выражение }> \; == \; <\text{ слагаемое }> | \\
 & <\text{ выражение }> \; ! = \; <\text{ слагаемое }> | \\
 & + <\text{ слагаемое }> | \\
 & - <\text{ слагаемое }> | \\
 & <\text{ слагаемое }> \\
<\text{ слагаемое }>\to & <\text{ слагаемое }> + <\text{ множитель }> | \\
 & <\text{ слагаемое }> - <\text{ множитель }> | \\
 & <\text{ множитель }> \\
<\text{ множитель }>\to & <\text{ множитель }> * <\text{ эл.выр. }> | \\
 & <\text{ множитель }> / <\text{ эл.выр. }> | \\
 & <\text{ эл.выр. }> \\
<\text{ эл.выр. }>\to & <\text{ имя }> | <\text{ константа }> | \\
 & <\text{ вызов функции }> |(<\text{ выражение }>)
\end{array}
$$

Осталось определить структуру констант и правила формирования вызова функций. Если в качестве фактических параметров при вызове функции можно использовать любое выражение, а функция может иметь любое число параметров, в частности, параметров может не быть совсем, то вызов функции определяется следующими правилами:

$$
\begin{array}{ll}
<\text{ вызов функции }>\to & <\text{ идентификатор }> (<\text{ параметры }>)| \\
 & <\text{ идентификатор }> () \\
<\text{ параметры }>\to & <\text{ параметры }> , <\text{ выражение }> | <\text{ выражение }>
\end{array}
$$

Теперь дадим определение констант и идентификаторов. Идентификатором является произвольная последовательность букв и цифр, начинающаяся с буквы. Целые константы, как уже отмечалось ранее, рассматриваются только в беззнаковом варианте. Целая константа — это последовательность цифр. Константа вещественная существует в двух вариантах — с точкой и в экспоненциальной форме. Проще всего последовательно строить соответствующие конструкции, используя уже имеющиеся:

$$
\begin{array}{ll}
<\text{ идентификатор }>\to & <\text{ буква }><\text{ окончание }> \\
<\text{ окончание }>\to & <\text{ окончание }><\text{ буква }> | \\
 & <\text{ окончание }><\text{ цифра }> |\varepsilon \\
<\text{ константа }>\to & <\text{ конст.целая }> | <\text{ конст.веществ. }> | \\
 & <\text{ конст.экспон. }> | <\text{ конст.символьн. }> \\
<\text{ конст.целая }>\to & <\text{ конст.целая }><\text{ цифра }> | <\text{ цифра }> \\
<\text{ конст.веществ. }>\to & <\text{ конст.целая }> . <\text{ конст.целая }> |. <\text{ конст.целая }> | \\
 & <\text{ конст.целая }> . \\
<\text{ конст.экспонен. }>\to & <\text{ конст.целая }><\text{ экспонента }> | \\
 & <\text{ конст.веществ. }><\text{ экспонента }>
\end{array}
$$

$$
\begin{array}{ll}
<\text{экспонента}> \to & \mathbf{E} <\text{знак}><\text{конст.целая}> \\
<\text{знак}> \to & + | - | \varepsilon \\
<\text{цифра}> \to & \mathbf{0}|\mathbf{1}|\mathbf{2}|\mathbf{3}|\mathbf{4}|\mathbf{5}|\mathbf{6}|\mathbf{7}|\mathbf{8}|\mathbf{9} \\
<\text{буква}> \to & \mathbf{a}|\mathbf{b}|\mathbf{c}|...|\mathbf{z} \\
<\text{конст.символьн.}> \to & '' <\text{символы}> '' \\
<\text{символы}> \to & <\text{символы}><\text{один символ}> | \varepsilon \\
<\text{один символ}> \to & <\text{буква}> | <\text{цифра}> | ...
\end{array}
$$

Грамматика языка построена. На лексический уровень вынесем следующие элементы:

1) идентификаторы; отметим, что в правилах описания идентификатора мы указали буквы только нижнего регистра — это означает, что при реализации сканера нам придется выбирать один из двух вариантов: либо буквы верхнего и нижнего регистров неразличимы, либо можно использовать только буквы нижнего регистра;

2) символьные константы, константы целые, константы вещественные с точкой, константы в экспоненциальной форме;

3) ключевые слова **if**, **for**, **function**, **var**, **else**, **script**, **javascript**, **language**;

4) специальные знаки: точка, точка с запятой запятая, круглые и фигурные скобки;

5) знаки операций $<=$, $>$, $>=$, $==$, $!=$, $+$, $-$, $*$, $/$, $\%$;

6) знаки начала и завершения тегов $<$, $</$, $>$, $<!--$, $-->$. Два знака из этих знаков–ограничителей совпадают по написанию со знаками операций отношения. Это означает, что на лексическом уровне данные знаки неразличимы.

Все выделенные символы являются терминальными в грамматике, описывающей синтаксический уровень.

3.8.3 Варианты заданий

В каждом задании описывается некоторый очень усеченный вариант известных языков программирования Java, C++ и Паскаль. В задании указывается:

1) структура программы,
2) типы данных, которые могут использоваться в программе,
3) допустимые операции над этими данными,
4) операторы,
5) операции и операнды, из которых строятся выражения,
6) все виды констант, которые могут использоваться в выражениях.

Обратите особое внимание на следующие требования к языку:

1) во всех заданиях предполагается использование составного и пустого оператора,
2) всегда допускается описание глобальных данных,
3) все перечисленные элементы языка должны использоваться в программе (например, если разрешается описание функций, то, безусловно, в перечень операторов Вам необходимо включить вызовы функций).

1. **Программа**: главная программа языка C++. Допускается описание структур. **Типы данных:** int, double. **Операции**: арифметические и сравнения. **Операторы**: присваивания и if. **Операнды**: простые переменные, элементы записей и константы. **Константы**: целые в 10 с/с , вещественные с фиксированной точкой.

2. **Программа**: главная программа языка C++. Допускается описание функций с параметрами, функции возвращают void. **Типы данных:** short int и long int. **Операции**: арифметические, сравнения. **Операторы**: присваивания и while. **Операнды**: простые переменные и константы. **Константы**: целые в 10 с/с и 16 с/с .

3. **Программа**: главная программа языка C++. Допускается описание функций без параметров допустимых в программе типов. **Типы данных:** float, char. **Операции**: арифметические и сравнения. **Операторы**: присваивания и do - while. **Операнды**: простые переменные, элементы одномерных массивов и константы. **Константы**: строковые, символьные и целые в 10 с/с .

4. **Программа**: главная программа языка C++. Допускается описание массивов как типов. **Типы данных:** int, int64. **Операции**: сравнения, логические, битовые. **Операторы**: присваивания и for простейшей структуры (цикл по одной переменной с заданным шагом). **Операнды**: простые переменные, элементы массивов и константы. **Константы**: целые в 10 с/с и 16 с/с .

5. **Программа**: главная функция языка C++. **Типы данных:** int, double. **Операции**: унарные и бинарные арифметические, сравнения. **Операторы**: присваивания и if. **Операнды**: простые переменные, элементы массивов и константы. **Константы**: целые, символьные, строковые.

6. **Программа**: главная программа языка C++. Допускается описание функций с параметрами. Функции возвращают значение. **Типы данных:** int (знаковые и беззнаковые). **Операции**: арифметические, сдвига, сравнения. **Операторы**: присваивания и while. **Операнды**: простые переменные, константы. **Константы**: все целые и символьные.

7. **Программа**: главная программа языка C++. Допускается описание функций с параметрами. Функции возвращают значение. **Типы данных:** int, double **Операции**: арифметические, сдвига, сравнения. **Операторы**: присваивания и if. **Операнды**: простые переменные, константы. **Константы**: все целые и символьные.

8. **Программа**: главная программа языка C++. Допускается описание функций без параметров. Функции возвращают значения допустимых в программе типов. **Типы данных**: double, char. **Операции**: арифметические и сравнения. **Операторы**: присваивания и while. **Операнды**: простые переменные, элементы одномерных массивов и константы. **Константы**: строковые, символьные и целые в 10 с/с .

9. **Программа**: главная программа языка C++. Допускается описание функций с параметрами, Функции возвращают тип void. **Типы данных:** int, char. **Операции**: простейшие арифметические и битовые. **Операторы**: присваивания и for. **Операнды**: простые переменные и константы. **Константы**: символьные и целые.

10. **Программа**: главная программа языка C++. Допускается описание функций без параметров типа void. **Типы данных:** int (short, long). **Операции**: арифметические, сдвига, сравнения. **Операторы**: присваивания и if. **Операнды**: простые переменные и именованные константы. **Константы**: целые в 10 с/с и 16 с/с .

11. **Программа**: главная программа языка C++. Допускается описание массивов как типов. **Типы данных:** int, пользовательские типы. **Операции**: сравненния, логические, битовые, адресные. **Операторы**: присваивания и if. **Операнды**: простые переменные, элементы массивов и именованные константы. **Константы**: целые в 10 с/с и 16 с/с .

12. **Программа**: главная функция языка C++. Допускается описание структур. **Типы данных:** int, char. **Операции**: унарные и бинарные арифметические, сравне-

ния. **Операторы**: присваивания и do-while. **Операнды**: простые переменные, элементы структур. **Константы**: целые, символьные, строковые.

13. **Программа**: главная программа языка С++. Допускается описание структур. **Типы данных:** short int, double. **Операции**: арифметические и сравнения. **Операторы**: присваивания и if. **Операнды**: простые переменные, элементы структур и именованные константы. **Константы**: целые в 10 с/с, вещественные с фиксированной точкой.

14. **Программа**: главная программа языка С++. Допускается описание struct. **Типы данных:** int, double, пользовательские типы. **Операции**: сравнения, логические, битовые, адресные. **Операторы**: присваивания и while. **Операнды**: простые переменные, элементы структур и константы. **Константы**: целые в 10 с/с и 16 с/с.

15. **Программа**: главная функция языка С++. Допускается описание массивов в конструкции typedef. **Типы данных:** int, double. **Операции**: унарные и бинарные арифметические, сравнения. **Операторы**: присваивания и switch. **Операнды**: простые переменные, элементы массивов. **Константы**: целые, символьные, строковые.

16. **Программа**: главная программа языка С++. Допускается описание struct. **Типы данных:** short int, double. **Операции**: арифметические и сравнения. **Операторы**: присваивания и if. **Операнды**: простые переменные, элементы структур и константы. **Константы**: целые в 10 с/с, вещественные в экспоненциальной форме.

17. **Программа**: главная программа языка С++. Допускается описание функций. Функции имеют параметры. **Типы данных:** int, boolean. **Операции**: арифметические и логические. **Операторы**: присваивания и if. **Операнды**: простые переменные, константы. **Константы**: целые в 10 с/с, логические.

18. **Программа**: главная программа языка С++.. Допускается классов. Функции не имеют параметров. **Типы данных:** float, int. **Операции**: все арифметические, сравнения. **Операторы**: присваивания и for. **Операнды**: простые переменные, константы. **Константы**: целые в 10 с/с.

19. **Программа**: главная программа языка С++. Допускается описание функций с параметрами. **Типы данных:** int, boolean. **Операции**: простейшие арифметические, сравнения и логические. **Операторы**: присваивания и while. **Операнды**: простые переменные, константы. **Константы**: целые в 10 с/с, целые в 16 с/с, логические.

20. **Программа**: класс Main языка Java. Допускается описание внутренних классов. Функции имеют параметры. **Типы данных:** int, boolean. **Операции**: арифметические и логические. **Операторы**: присваивания и if. **Операнды**: простые переменные, константы. **Константы**: целые в 10 с/с, логические.

21. **Программа**: класс Main языка Java. Допускается описание внутренних классов. Функции не имеют параметров. **Типы данных:** float, int. **Операции**: все арифметические, сравнения. **Операторы**: присваивания и for. **Операнды**: простые переменные, константы. **Константы**: целые в 10 с/с.

22. **Программа**: класс Main языка Java. Допускается описание функций с параметрами. **Типы данных:** int, boolean. **Операции**: простейшие арифметические, сравнения и логические. **Операторы**: присваивания и while. **Операнды**: простые переменные, константы. **Константы**: целые в 10 с/с, целые в 16 с/с, логические.

23. **Программа**: класс Main языка Java. Допускается описание массивов любой размерности и вложенных классов. **Типы данных:** int, float. **Операции**: арифметические и сравнения. **Операторы**: присваивания и switch. **Операнды**: простые

переменные, элементы массивов и константы. **Константы**: целые в 10 с/с и вещественные в экспоненциальной форме.

24. **Программа**: класс Main языка Java. Допускается описание внутренних классов. Функции имеют параметры. **Типы данных:** double, boolean. **Операции**: простейшие арифметические и логические. **Операторы**: присваивания и if. **Операнды**: простые переменные, константы. **Константы**: целые в 10 с/с, логические.

25. **Программа**: главная программа языка Паскаль. Допускается описание процедур без параметров. **Типы данных:** integer (длинные и короткие), boolean. **Операции**: арифметические, логические, сравнения. **Операторы**: присваивания и if. **Операнды**: простые переменные и константы. **Константы**: целые в 10 с/с и 16 с/с , true, false.

26. **Программа**: главная программа языка Паскаль. Допускаются функции без параметров. **Типы данных:** integer, set of. **Операции**: унарные и бинарные арифметические, сравнения, над множествами. **Операторы**: присваивания и do-while. **Операнды**: простые переменные, множества и константы. **Константы**: целые.

27. **Программа**: главная программа языка Паскаль. Допускается описание процедур с параметрами. **Типы данных:** integer, char. **Операции**: простейшие арифметические. **Операторы**: присваивания и for. **Операнды**: простые переменные и константы. **Константы**: символьные и целые.

28. **Программа**: главная программа языка Паскаль. Допускается описание записей. **Типы данных:** integer (длинные и короткие). **Операции**: сравнения и арифметические. **Операторы**: присваивания и do-while. **Операнды**: простые переменные, константы. **Константы**: целые в 10 с/с и 16 с/с , в том числе и long.

29. **Программа**: главная программа языка Паскаль. Допускается описание массивов как типов, а также процедур без параметров **Типы данных:** integer, boolean. **Операции**: сравнения, логические и арифметические. **Операторы**: присваивания и if. **Операнды**: простые переменные, элементы массивов и константы. **Константы**: целые в 10 с/с и 16 с/с .

30. **Программа**: главная программа языка Паскаль. Допускается описание функций без параметров допустимых в программе типов. Разрешаются рекурсивные вызовы. **Типы данных:** integer, real. **Операции**: арифметические, логические, сравнения. **Операторы**: присваивания и if. **Операнды**: простые переменные и константы. **Константы**: все арифметические.

31. **Программа**: главная программа языка Паскаль. Допускается описание функций без параметров допустимых в программе типов. **Типы данных:** real, char. **Операции**: арифметические и сравнения. **Операторы**: присваивания и while. **Операнды**: простые переменные, элементы одномерных массивов и константы. **Константы**: вещественные, символьные и целые в 10 с/с .

32. **Программа**: главная программа языка Паскаль. Допускается описание записей. **Типы данных:** integer, real. **Операции**: арифметические и сравнения. **Операторы**: присваивания и if. **Операнды**: простые переменные, элементы записей и константы. **Константы**: целые в 10 с/с и вещественные с фиксированной точкой.

Глава 4

ЛЕКСИЧЕСКИЙ АНАЛИЗ

4.1 Формальные параметры функции сканера

Лексический анализ является первым этапом процесса компиляции. На этом этапе терминальные символы, составляющие входную программу, группируются в отдельные лексические элементы, называемые лексемами. Сразу отметим, что сканер — это процедура или функция, которая за одно обращение выделяет только одну очередную лексему. Никакая синтаксическая информация сканеру не доступна, синтаксис языка не должен учитываться при программировании сканера. Именно по этим причинам сканер должен выделять только беззнаковые константы, т.к. знак операции и знак перед константой могут быть опознаны только на синтаксическом уровне. Например, в выражении языка C++

```
y=-3-2-(+6+7);
```

для программиста очевидно наличие трех знаков операции и двух констант со знаком, которые стоят в начале выражения и после открывающейся скобки. Сканер выделит пять знаков операции и четыре беззнаковых константы. Анализ структуры выражения, построенного из знаков и операндов, — задача синтаксического анализатора.

Введем следующие определения:

```
#define MaxText 10000       //максимальная длина текста ИМ
#define MaxLex  20          //максимальная длина лексемы
typedef char IM[MaxText];   // текст ИМ
typedef char LEX[MaxLex];   // лексема.
```

Тогда исходные данные сканера — это текст исходного модуля и указатель очередной анализируемой позиции в этом тексте:

```
IM t;    // исходный модуль
int uk;  // указатель текущей позиции в ИМ.
```

Как правило, текст t и указатель uk являются глобальными данными для функции сканера, поскольку они одни и те же в любой точке программы компилятора. Обычно дополнительно используются два указателя — указатель строки и позиции в строке

```
int line, pos; //строка, позиция
```

которые позволяют визуально указать пользователю на ошибку в программе. Эти данные также будем считать глобальными.

Рассмотрим теперь выходные данные сканера. Обычно с лексемой связывают лексическую структуру, содержащую следующую информацию:

- изображение лексемы,
- тип лексемы,
- некоторые данные о лексеме, например, адрес таблицы информации, где хранятся сведения о лексеме.

Первые компоненты являются результатом работы сканера и используются анализатором, последняя — семантическими подпрограммами и генератором объектного кода. Следовательно, сканер должен возвращать два значения:

```
int typ; // тип лексемы
LEX l;   // изображение лексемы.
```

В качестве типа мы выбрали значение типа *int* — тем самым фактически пронумеровав все возможные типы лексем в языке программирования. Реализуем сканер в виде функции языка C++, которая возвращает в качестве значения тип лексемы и имеет один параметр — изображение лексемы:

```
int Scaner(LEX   l)
{int typ; // тип лексемы
...
return typ; }  // конец Scaner
```

Для того, чтобы проиллюстрировать необходимость одновременного использования как изображения лексемы, так и ее типа, рассмотрим фрагмент программы на языке C++:

```
  a=b*2+3;
  alpha= betta*1234+7777;
```

На синтаксическом уровне — это два оператора одной и той же конструкции. Чтобы выполнить грамматический разбор этой конструкции, синтаксическому анализатору нужны типы лексем ”идентификатор” и ”константа”. В то же время для генерации объектного кода нужны изображения лексем, в противном случае оба оператора получат один и тот же перевод в ассемблерную программу.

4.2 Таблица лексических единиц

В предшествующем параграфе мы обсудили входные и выходные данные функции сканера. Перейдем теперь к реализации тела этой функции. Прежде, чем писать программу, нужно выяснить, какие именно лексические единицы будет выделять наш сканер. В этом нам поможет построение таблицы лексем. Построим эту таблицу следующим образом. Каждая лексема занимает одну строку таблицы. Колонки таблицы определяют

- название лексемы,
- тип лексемы,

• символ–ограничитель,

• дополнительную информацию, необходимую для выделения лексемы.

Построенная таблица дополняется двумя специальными типами — это ошибочный символ и конец исходного модуля. Как уже отмечалась ранее, в качестве типа выберем целое число — номер типа. При программной реализации, чтобы избежать ошибок при сравнении типов, лучше воспользоваться символьным обозначением каждого внесенного в таблицу типа. Это проще всего сделать одним из двух способов:

• с использованием перечисляемых данных типа *enum*,

• с помощью макроопределения в $\natural define$.

Для простоты реализации ключевым словам присваивают типы в виде возрастающей последовательности целых чисел (обычно, начиная с единицы), а типы лексем формируются в программе в виде встроенной таблицы, например,

```
#define KeyInt       1
#define KeyFloat     2
...
#define ConstInt    10
#define ConstFloat  11
...

#define Plus        20
#define Minus       21
...
#define TypeErr    400
#define TypeEnd    500
```

Лексика языков программирования обычно проста и описывается регулярными выражениями, поэтому после того, как в грамматике языка программирования выделены синтаксический и лексический уровни, можно построить конечный автомат, соответствующий лексике языка.

Из теории автоматов известно, что для произвольного конечного автомата можно найти эквивалентный минимальный автомат, исключая все недостижимые состояния и склеивая лишние состояния. Лишние состояния определяются с помощью разделения всех достижимых состояний на классы эквивалентности так, что каждый класс содержит неразличимые состояния. Таким образом можно сократить объем автомата, а тем самым и программу, моделирующую поведение этого автомата.

Другой не менее важной проблемой, чем проблема минимизации автомата, при программировании сканера является проблема детерминированности конечного автомата, на основе которого определяется лексика языка программирования. Известно, что для произвольного конечного автомата существует эквивалентный детерминированный, который строится на основе весьма простого алгоритма. Очень часто в процессе детерминизации произвольного конечного автомата появляется много новых состояний и автомат существенно разрастается по сравнению с исходным недетерминированным. Как правило, лексика языков программирования достаточно проста, поэтому при детерминизации автомата, описывающего лексику, такое разрастание не происходит. Объясняется этот факт тем, что большинство недетерминированностей разрешается или за счет объединения нескольких фрагментов автомата в один фрагмент или за счет встраивания одного фрагмента в другой. В качестве примера можно привести десятичные константы — целые, с фиксированной точной и в экспоненциальной форме. Объединение отдельных фрагментов автомата, соответствующих указанным константам, просто приведет к слиянию этих фрагментов, что фактически отразится только на переносе заключительных состояний в самый сложный результирующий фрагмент (см. рис. 4.1 и рис. 4.2).

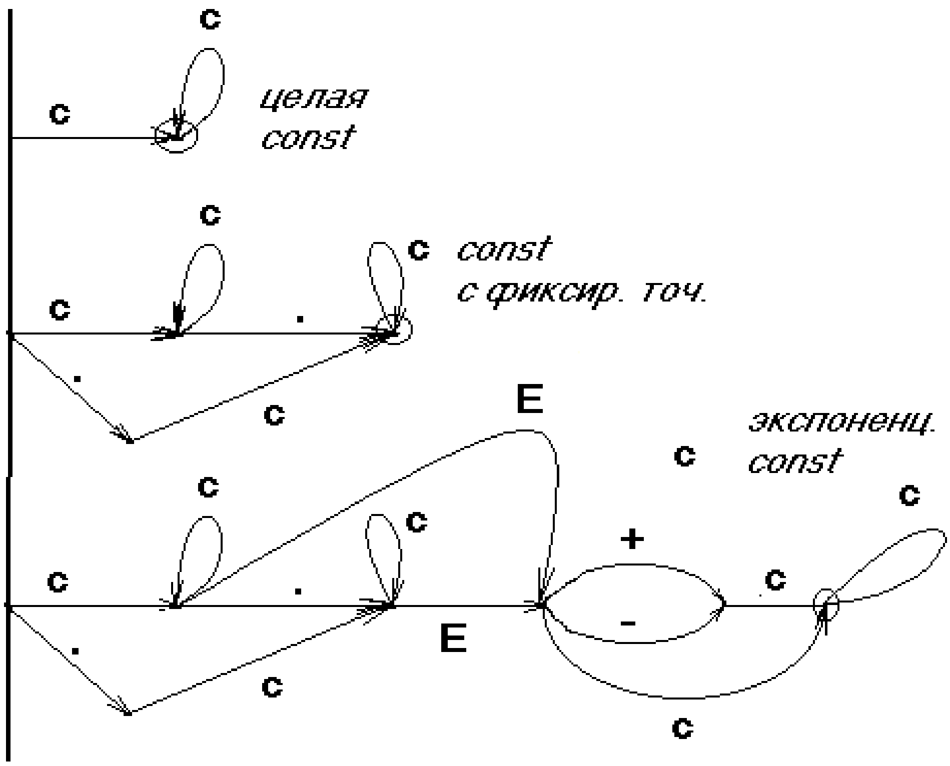


Рис. 4.1: Недетерминированный конечный автомат, представляющий константы

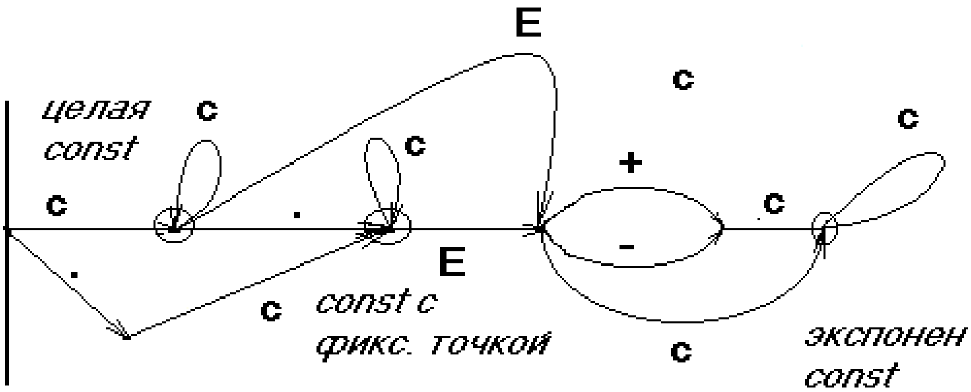


Рис. 4.2: Детерминированный конечный автомат, представляющий константы

На рисунке символ "c" — это изображение произвольной десятичной цифры, а начальное состояние представлено в виде начальной вертикальной линии. Способ изображения начального состояния в виде одной вертикальной линии весьма часто используется при изображении автоматов, у которых из начального состояния выходит много дуг.

Более сложная ситуация возникает при построении детерминированного конечного автомата, распознающего ключевые слова и идентификаторы. Строго говоря, для выделения множества тех и только тех последовательностей латинских букв, которые являются ключевыми словами, нужно построить конечный автомат, который представляет собой множество параллельных ветвей между начальным и заключительным состоянием. Каждая такая ветвь — линейная последовательность дуг, соответствующих буквам ключевого слова. Для реальных языков программирования такой автомат совершенно непригоден для практической реализации сканера. Поэтому поступают иначе. Поскольку множество ключевых слов по своей конструкции удовлетворяют правилам построения идентификаторов, конечный автомат, описывающий лексику языка программирования, содержит только фрагмент конструкции для регулярного выражения идентификатора. А после выделения идентификатора в заключительном состоянии автомата выделенная лексема проверяется на ее совпадение с таблицей ключевых слов.

4.3 Программирование сканера

Итак, лексика языков программирования описывается регулярными выражениями, а, следовательно, конечными автоматами. Перед программированием этот конечный автомат необходимо преобразовать к эквивалентной детерминированной форме. Таким образом, программировать сканер можно на основе программной модели детерминированного конечного автомата. Существуют *два способа программирования конечного автомата: явный и неявный.*

При *явном способе* в программе имеется таблица переходов конечного автомата и переменная, значением которой является номер состояния этого автомата. Переходы выполняются в соответствии с таблицей переходов. Способ универсален, т.к. изменение лексики осуществляется простым изменением таблицы переходов, однако, он обладает низким быстродействием из-за необходимости обработки таблицы. Существуют различные способы представления таблиц, увеличивающие скорость их обработки.

Рассмотрим *неявный способ* программирования конечного автомата, при котором переменная — состояние автомата — отсутствует, а каждому переходу конечного автомата ставится в соответствие уникальный фрагмент программы, реализующий действия из таблицы переходов. Таким образом, переход из состояния в состояние моделируется переходом от одного фрагмента программы к другому. А тот факт, что моделируемый конечный автомат находится в заданном состоянии, реализуется тем, что моделирующая программа исполняет часть кода, которая соответствует этому состоянию. Программа сканера, запрограммированного неявным способом, обладает высоким быстродействием, но является уникальной для каждого языка программирования и требует корректировки при изменении лексики языка.

Явный или неявный способ программирования выбирается исходя из назначения сканера и требований к его универсальности или быстродействию.

Следует также отметить некоторые особенности чтения символов исходной цепочки. Если бы множество символов исходного модуля непосредственно служило входом некоторого конечного автомата, то в соответствии с формальным определением автомат имел бы очень большую таблицу переходов (соответственно, и большую программу сканера), т.к. только одни цифры и латинские буквы одного регистра образуют множество из 36 символов. Поэтому при неявном способе программирования конечного автомата проверка на букву или цифру осуществляется по диапазону изменения значения: например,

```
"if ( (t[uk]>='a') && (t[uk]<='z') ) " ---
```

маленькая латинская буква. При явном способе программирования конечного автомата дополнительно используется так называемый *транслитератор* — автомат, единственная задача которого — сократить входное множество до приемлемых размеров. Зависимость между входом и выходом транслитератора можно определить с помощью таблицы, которая содержит перевод каждого ASCII–символа в его тип. Аналогично решается проблема выделения ключевых слов языка программирования. Если бы автомат распознавал отдельно каждое ключевое слово, то таблица переходов такого автомата имела бы сотни состояний при наличии десятков ключевых слов. Поэтому в качестве регулярного выражения для ключевого слова выбирается регулярное выражение идентификатора, и только после выделения идентификатора осуществляется проверка выделенной лексемы на совпадение ее с некоторым элементом в таблице ключевых слов. Поэтому при программировании обычно выбирают в качестве значения типа ключевого слова его индекс в таблице ключевых слов.

Сразу следует предостеречь от попытки использования вспомогательных функций–транслитераторов для проверки типа очередного символа при неявном способе программирования сканера. Все преимущества в скорости работы такого сканера будут ликвидированы за счет многократного использования вызовов таких функций. В программе сканера имеется не так много участков, в которых проверяется символ одного и того же типа. Если таких участков все же много, лучше воспользоваться inline–функциями или макроопределениями:

```
#define LetterSmall     ( (t[uk]>='a') && (t[uk]<='z') )
#define LetterBig       ( (t[uk]>='A') && (t[uk]<='Z') )
#define Number          ( (t[uk]>='0') && (t[uk]<='9') )
...
if (LetterSmall || LetterBig || Number)  ...
```

Обычно на лексическом уровне некоторые символы игнорируются. К таким символам относятся пробелы, символы перевода строки, комментарии и т.п. Эти символы в начале работы сканера необходимо пропустить, увеличивая в процессе такого пропуска указатель *uk*. Если для программы выдачи ошибки отслеживается значение указателя строки *line* и значение позиции в строке *pos*, то их в процессе пропуска незначащих символов необходимо изменять соответствующим образом.

Для того, чтобы отследить конец исходного модуля, запишем в конец исходного модуля специальный маркер конца, например, знак "♮" или "\0". При достижении этого знака нельзя увеличивать значение указателя *uk*. Сохранение неизменным значения переменной *uk* в конце исходного модуля необходимо для того, чтобы синтаксический анализатор не мог перейти к анализу текста за концом исходного модуля.

Таким образом, программа сканера имеет вид:

```
int Scaner(LEX  l)  {
   int typ;             // тип лексемы
   int i;         // текущая длина лексемы
   for (i=0;i<MaxLex;i++) l[i]=0;  //очистили поле лексемы
   i=0;                 // лексема заполняется с позиции i
   while ( (t[uk]==' ') || (t[uk]=='\n') || ...)   uk++;
                        // пропуск незначащих элементов
   if (t[uk]=='\0')  {
        l[0]=t[uk];
        return <тип конца ИМ>;
   }
   else  {
     <модель КА>
```

```
      ...
    return <тип лексемы>;
  }
} // конец Scaner
```

Программа конечного автомата представляет собой последовательность действий для каждого состояния. Каждому состоянию поставим в соответствие метку, тогда фрагмент программы сканера, представляющий модель конечного автомата, имеет вид:

```
Label_1: <действия в состоянии 1>
...
Label_n: <действия в состоянии n>
```

Поскольку конечный автомат начинает работу из начального состояния, программа сканера должна начинаться с фрагмента, соответствующего этому состоянию. Поэтому в последовательности фрагментов первым указывается начальное состояние, последующий порядок состояний безразличен.

Программа для каждого состояния зависит от типа этого состояния (начальное, заключительное, промежуточное) и от переходов из данного состояния. Тот факт, что переход в текущее состояние был выполнен из некоторого другого состояния, никак не отражается на программе, т.к. по определению конечный автомат — это автомат без памяти. Рассмотрим всевозможные варианты состояний.

4.3.1 Вариант 1 — простое состояние

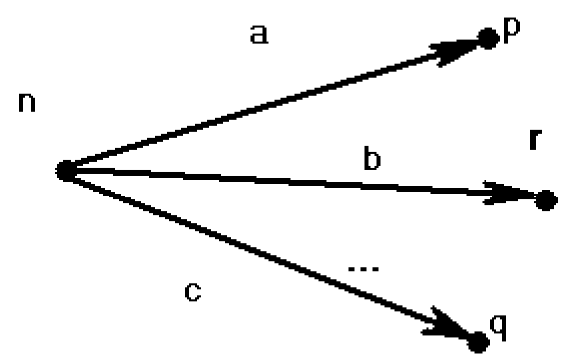


Рис. 4.3: Программируемое состояние автомата — не начальное и не заключительное

Если текущее состояние n - не начальное и не заключительное (см. рис. 4.3), то нужно только проверить правильность очередного символа и выполнить переход в новое состояние. Очередной правильный символ предварительно нужно приписать в конец формируемой лексемы l. Любой другой символ означает ошибку в лексеме. Тогда программа состояния имеет вид:

```
Label_n: if (t[uk]=='a')
               { l[i]=t[uk]; i++; uk++; goto Label_p;}
         else  if (t[uk]=='b')
               { l[i]=t[uk]; i++; uk++; goto Label_r;}
```

```
         else ...
         else  if (t[uk]=='c')
                   { l[i]=t[uk]; i++; uk++; goto Label_q;}
         else { PrintError(...);  // ошибка
                  return <ошибочный тип>;
               }
```

Очевидно, что последовательность операторов

```
l[i]=t[uk]; i++; uk++;
```

проще заменить на один оператор

```
l[i++]=t[uk++];
```

Программа выдачи ошибки $PrintError(...)$ обычно строится либо как оператор *switch* по номерам ошибки (тогда ее параметр — номер ошибки), либо как оператор печати текста, который является параметром (тогда параметр — текст сообщения об ошибке). Кроме того, процедура $PrintError(...)$ должна указывать местоположение ошибки или хотя бы номер строки и изображение неверного символа. Именно для такого сообщения используются указатели *line* и *pos*. Сканер, как правило, выдает два типа ошибок:

- недопустимый символ,
- ошибка в структуре лексемы.

Очень часто при конструировании лексем используется операция итерации или усеченной итерации. Например, идентификатор задается регулярным выражением

$$b(b \cup c)^*,$$

а десятичная константа с фиксированной точной имеет вид

$$c^*.c^+ \cup c^+.c^*.$$

Здесь знаки b и c обозначают одну букву и одну цифру соответственно. Наличие таких определений приводит к появлению "петель" в конечном автомате. Очевидно, что для их реализации можно использовать как универсальный метод с оператором *goto*, предложенный выше, так и использование цикла *while*. Например, состояние выделения окончания идентификатора, состоящего из последовательности букв нижнего и верхнего регистров и цифр, можно реализовать в виде цикла

```
Label_n: while (  ((t[uk]>='a') && (t[uk]<='z')) ||
                  ((t[uk]>='A') && (t[uk]<='Z')) ||
                  ((t[uk]>='0') && (t[uk]<='9')) ) l[i++]=t[uk++];
```

При сканировании лексем должна быть предусмотрена возможность отсечения "хвоста" длинных объектов. Для этого в состояниях, находящихся в пределах циклических участков конечного автомата, необходимо предусматривать проверку переменной i на превышение допустимой длины лексемы $MaxLex$. В этом случае "хвост" лексемы отсекается и (в зависимости от соглашения о длине лексем в языке программирования) выдается либо ошибка, либо предупреждение.

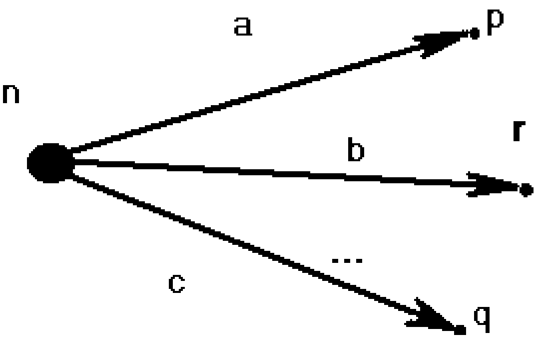


Рис. 4.4: Программируемое состояние автомата — начальное

4.3.2 Вариант 2 — начальное состояние

Расмотрим теперь вариант, когда состояние n — начальное (см. рис. 4.4). Программа практически совпадает с предшествующим вариантом, только при обнаружении ошибки необходимо увеличить на единицу указатель uk, чтобы избежать зацикливания на выделении одного и того же неверного символа при последующих обращениях к сканеру:

```
Label_n: if (t[uk]=='a')
                  { l[i++]=t[uk++]; goto Label_p;}
         else  if (t[uk]=='b')
                  { l[i++]=t[uk++]; goto Label_r;}
         else ...
         else  if (t[uk]=='c')
                  { l[i++]=t[uk++]; goto Label_q;}
         else { PrintError(...);  // ошибка
                uk++;
                return <ошибочный тип>;
              }
```

Еще раз напоминаем, что фрагмент программы для начального состояния должен стоять первым в сканере, т.к. конечный автомат начинает работу из начального состояния.

4.3.3 Вариант 3 — тупиковое заключительное состояние

Заключительное состояние, из которого нет переходов (см. рис. 4.5), можно запрограммировать как оператор возврата из программы сканера. С учетом того, что сканер должен возвращать тип выделенной лексемы, в состоянии n могут быть реализованы дополнительные действия по определению типа лексемы или контролю ее структуры. Например, ключевые слова проще сканировать как обычные идентификаторы, а потом по совпадению с таблицей ключевых слов сделать вывод о том, что отсканировано ключевое слово. В некоторых языках программирования десятичные и восьмеричные константы отличаются первой цифрой — восьмеричные константы начинаются со знака нуля. Десятичные и восьмеричные константы можно сканировать в соответствии с регулярным выражением c^+, а затем проверить выделенную

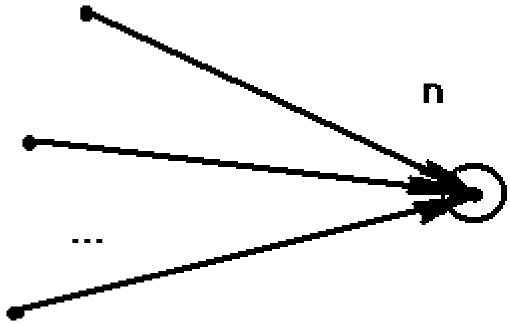


Рис. 4.5: Программируемое состояние автомата — тупиковое заключительное

константу на ее совпадение со структурой восьмеричной константы. Таким образом, программа тупикового заключительного состояния имеет вид:

```
Label_n: <возможные действия по определению типа лексемы>
      return <тип лексемы, соответствующей состоянию>;
```

В большинстве случаев проще эти действия перенести в состояние, из которого был переход на данное состояние. Имеет смысл оставить отдельную выделенную реализацию состояния только в том случае, когда дополнительные действия нетривиальны, а переход в состояние n выполняется из нескольких различных состояний. Анализ лексики большинства известных языков программирования показывает, что такая ситуация на практике встречается исключительно редко.

4.3.4 Вариант 4 — заключительное состояние с переходами

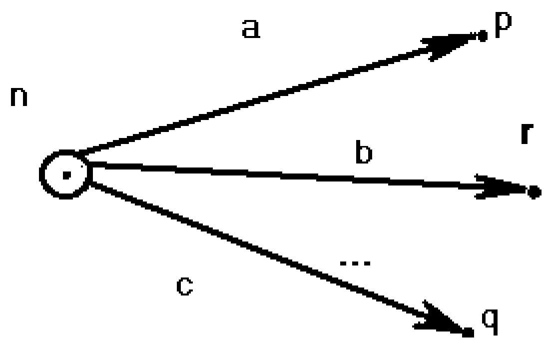


Рис. 4.6: Программируемое состояние автомата — заключительное с переходами

Последним рассмотрим вариант, когда состояние n — заключительное (см. рис. 4.6). Программа отличается от программы для первого варианта тем, что вместо выдачи ошибки ставится определение типа лексемы и выход из программы сканера:

```
Label_n: if (t[uk]=='a')
                  { l[i++]=t[uk++]; goto Label_p;}
```

```
else  if (t[uk]=='b')
          { l[i++]=t[uk++]; goto Label_r;}
else ...
else  if (t[uk]=='c')
          { l[i++]=t[uk++]; goto Label_q;}
else { <дополнительные действия по определению типа>
       return <тип лексемы для состояния n>;
     }
```

4.4 Простой пример программы сканера

Построим сканер, выделяющий целые беззнаковые десятичные константы, идентификаторы, состоящие из букв и цифр, знаки арифметических операций "+", "-", знаки операций сравнения "<", ">", "<=", ">=". Для простоты будем считать игнорируемыми символами только знаки пробела и перевода строки, а также не будем контролировать длину константы и идентификатора. Построим таблицу лексем:

лексема	тип лексемы	ограничитель лексемы
целая константа	type_const = 1	не цифра
идентификатор	type_ident = 2	не буква, не цифра
знаки "+", "-"	type_plus, type_minus = 3,4	любой знак
знак "<"	type_lt = 5	не знак "="
знак ">"	type_gt = 6	не знак "="
знак "<="	type_le= 7	любой знак
знак ">="	type_ge = 8	любой знак
ошибочный тип	type_error = 100	
конец текста	type_end = 200	

Построим конечный автомат, используя обозначения "*b*" и "*c*" для представления маленькой латинской буквы и цифры соответственно, знак "*bc*" — для представления одного символа, который может быть или буквой иди цифрой. Для наглядности начальное состояние представим в виде начальной вертикальной линии. Получим граф, представленный на рис. 4.7.

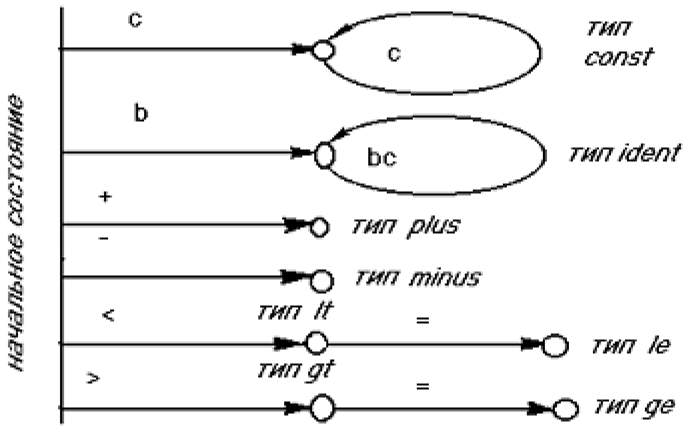


Рис. 4.7: Конечный автомат, представляющий лексику языка из примера к 4.4

В дальнейшем мы будем использовать класс *TScaner*, но в данный момент для иллюстрации принципов реализации нам достаточно использовать только функцию сканирования. Программа имеет вид:

```
#define type_const   1
#define type_ident   2
#define type_plus    3
#define type_minus   4
#define type_lt      5
#define type_gt      6
#define type_le      7
#define type_ge      8
#define type_error 100
#define type_end   200
int Scaner(LEX  l)
{
int typ;                                       // тип лексемы
int i;                                    // текущая длина лексемы
for (i=0;i<MaxLex;i++) l[i]=0;           //очистили поле лексемы
i=0;                // лексема заполняется с позиции i
while ( (t[uk]==' ') || (t[uk]=='\n') )      uk++;
                             // пропуск незначащих элементов
if (t[uk]=='\0'){l[0]='\0'; return type_end;}
   else
   {
N0: if ( (t[uk]<='9')&&(t[uk]>='0'))   {
        l[i++]=t[uk++]; goto N1;
        while ( (t[uk]<='9')&&(t[uk]>='0'))  // состояние N1
           l[i++]=t[uk++];
        return type_const;
    }
    else  if ( (t[uk]>='a')&&(t[uk]<='z'))  {
            l[i++]=t[uk++];
            while ( (t[uk]<='9')&&(t[uk]>='0') ||  // состояние N2
                    (t[uk]>='a')&&(t[uk]<='z'))   l[i++]=t[uk++];
            return type_ident;
          }
    else  if (t[uk]=='+')  {
            l[i++]=t[uk++]; return type_plus;
          }
    else  if (t[uk]=='-')  {
            l[i++]=t[uk++]; return type_minus;
          }
    else  if (t[uk]=='<')  {
             l[i++]=t[uk++];
             if (t[uk]=='=') { l[i++]=t[uk++]; return type_le; }
             else return type_lt;
          }
    else  if (t[uk]=='>')   {
```

```
            l[i++]=t[uk++];
            if (t[uk]=='=') { l[i++]=t[uk++]; return type_ge; }
            else return type_gt;
         }
    else { PrintError(10); // ошибка
          return type_error;
         }
} // конец Scaner
```

Здесь следует отметить, что состояния $N1$ и $N2$ реализованы в программе с помощью операторов *while*, а не *goto*, т.к. такая запись предпочтительнее.

4.5 Отладка программы сканера

Конечно, можно не выполнять отладку сканера как отдельной программы, а подождать того момента, пока будет написан синтаксический анализатор и попытаться выполнить отладку в комплексе. Теоретически, если Вы выполнили все проектные действия правильно, то программа должна работать без ошибок. Поскольку такое событие весьма маловероятно, лучше выполнить отладку сканера отдельно, а затем, зная, что сканер работает правильно, перейти к отладке синтаксического анализатора. Простейший способ отладки заключается в сканировании произвольного текста до тех пор, пока в читаемом исходном модуле имеются еще не проанализированные символы.

```
#include <stdio.h>
#include <string.h>

#define MAX_TEXT 10000
#define MAX_LEX    100
typedef char TypeLex[MAX_LEX];
char t[MAX_TEXT]; // исходный текст

int uk; // указатель текущей позиции в исходном тексте

FILE * in  = fopen("input.txt","r");

int Scaner(TypeLex l)
{
...
}


void GetData()
{
int i=0;
while(!feof(in))
    fscanf(in,"%c",&t[i++]);
t[i]='\0'; // приписываем  знак '\0' в конец текста
```

```
}


int main(void)
{
int type; Lex l;
GetData();   // ввести данные

do {
   type=Scaner(l);
   printf("%s  - тип  %d \n",l, type);
   } while(type!=type_end);
return 0;
}
```

Теперь следует сделать несколько замечаний относительно буфера для чтения текста. Если Вы уверены, что текст никогда не превысит по длине $MaxText$, то приведенный выше пример программы вполне работоспособен. Если текст может иметь существенно большую длину, то надо либо программно ограничить длину читаемого текста, либо воспользоваться динамическим выделением памяти в соответствии с длиной файла.

Предложенный выше вариант организации программы может быть использован для отладки программы сканера и вполне с этой точки зрения работоспособен. Однако Вам в дальнейшем придется спроектировать и запрограммировать компилятор или интерпретатор в целом, включая и синтаксический анализатор, и семантические программы контроля правильности, и действия по переводу или интерпретации. Если все соответствующие фрагменты Вы собираетесь включить в один модуль, Ваша попытка заранее обречена на неудачу. Большой программный комплекс должен состоять из целого набора модулей, объединенных в один проект. В нашем случае будем использовать следующие модули:

$defs.hpp$ — описания всех общих для компилятора типов данных и макросов;

Scaner.hpp — описание класса сканера; в список методов следует внести функции запоминания и восстановления указателя uk, функцию чтения исходного файла, а также функцию выдачи сообщения об ошибке;

Scaner.cpp — реализация методов класса функций, указанных в *Scaner.hpp*;

trans.cpp — главная программа транслятора. Пока в качестве такой программы будет выступать программа сканирования заданного текста и вывода отсканированных лексем и их типов.

Далее приведены тексты указанных модулей. В состав проекта входят модули *Scaner.cpp* и *trans.cpp*.

```
//**************************************************************
// модуль defs.hpp ---  общие типы данных и макроопределения
//**************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_TEXT 10000
```

```
#define MAX_LEX    100

#define KeyInt       1
#define KeyFloat     2
   ...
#define TypeEnd    500

typedef char TypeLex[MAX_LEX];



//**************************************************************
// модуль Scaner.hpp --- заголовки функций сканера
//**************************************************************
#include "defs.hpp"

void PutUK  (int i);
int  GetUK  (void);
void PrintError(int i);
int  Scaner (TypeLex l);
void GetData(void);


//**************************************************************
//модуль Scaner.cpp --- реализация функций сканера
//**************************************************************
#include "defs.hpp"
#include "Scaner.hpp"

char t[MAX_TEXT]; // исходный текст
int uk;           // указатель текущей позиции в исходном тексте
FILE * in;

void PutUK(int i){uk=i;}      //восстановить указатель

int GetUK(void){return uk;}   // запомнить указатель

void PrintError(int i)         // выдать сообщение об ошибке
{
   ...
}

int Scaner(TypeLex l)          // программа сканера
{
   ...
}

void GetData()              // ввод файла с исходным модулем
{
in  = fopen("input.txt","r");
```

```
int i=0;
while(!feof(in))
    fscanf(in,"%c",&t[i++]);
t[i]='\0';                  // приписываем знак '\0' в конец текста
fclose(in);
}

//*****************************************************************
// модуль trans.cpp ---  главная программа транслятора
//*****************************************************************
#include "defs.hpp"
#include "Scaner.hpp"

int main(void)
{
int type;
TypeLex l;   // локальные переменные типа и изображения лексемы
GetData();   // ввести данные
do {
   type=Scaner(l);
   printf("%d ---> %s\n",type,l);
   } while(type!=type_end);
return 0;
}
```

Особое внимание хотелось бы уделить парадигме программирования, которая используется при реализации программы компилятора. Идеология объектно–ориентированного программирования (ООП), безусловно, является основой промышленного проектирования программного обеспечения. В процессе всего нашего курса именно принцип ООП лежит в основе реализации. Фактически, приведенный выше пример является единственным примером реализации программы без использования классов, но с сохранением всех принципов ООП. Приведен этот пример с единственной целью: профессиональные программисты должны понимать два принципиальных момента:

• объектно–ориентированный подход *в проектировании* является фундаментальным принципом при разработке программного обеспечения, позволяет избежать многих ошибок в проектировании, создает возможность коллективной работы над проектом, является основой для повторного использования кода;

• объектно–ориентированный проект иногда можно *при реализации* представить без использования классов, сохраняя при этом объектно–ориентированную природу проекта.

Очевидно, что в соответствии со структуроя языка программирования, а, следовательно, и компилятора, должны быть спроектированы классы лексического уровня, синтаксического уровня, уровня контекстных условий, генерации кода, оптимизации. Фактически, приведенный выше код означает наличие класса

```
class TScaner{
private:
   TypeMod t;          // исходный текст
   int uk;             // указатель текущей позиции в исходном тексте
```

```
public:
   void PutUK  (int i);
   int  GetUK  (void);
   void PrintError(char *, char *);
   int  Scaner (TypeLex l);
   void GetData(char *);
   TScaner(char *);
   ~TScaner() {}
};
```

Предлагаемая структура программного комплекса для реализации компилятора является простой в исполнении, наглядной, основана на независимой реализации различных по смыслу программных модулей, следовательно, легко отлаживается. Иногда у программиста возникает желание в разных программных модулях использовать одни и те же данные. Как правило, это желание противоречит принципам объектно-ориентированного и модульного программирования, однако, в исключительных случаях сделать такой доступ к одним и тем же данным из разнах программных модулей все же требуется. Сделать это очень просто, если воспользоваться описателем *extern* для этих данных, а сами общие данные поместить еще в один программный модуль, который наравне с другими модулями входит в состав проекта.

Например, если некоторые данные

```
char Data[MaxData];
int a,b,c,d;
```

должны быть доступны в модуле $Mod1.cpp$ и $Mod2.cpp$, то описание данных формирует с специальном модуле *Data.cpp*, который вместе с модулями $Mod1.cpp$, $Mod2.cpp$ входит в один проект. А конструкция каждого из этих модулей имеет вид:

```
// модуль Data.cpp - описание данных
char Data[MaxData];
int a,b,c,d;
// конец модуля Data.cpp

// модуль Mod1.cpp, который использует внешние данные
extern  char Data[MaxData];
extern  int a,b,c,d;
   ...
// конец модуля   Mod1.cpp

// модуль Mod2.cpp, который использует внешние данные
extern  char Data[MaxData];
extern  int a,b,c,d;
   ...
// конец модуля   Mod2.cpp
```

Практика программирования показывает, что, как правило, такими внешними данными могут быть только какие-то переменные, значения которых определяются в результате чтения данных из конфигурационного файла. Во всех остальных случаях использование внешних данных вряд ли разумно. *При реализации компилятора в рамках данного курса постарайтесь не использовать внешние данные.*

4.6 Контрольные вопросы к разделу 2

1. Перечислите входные и выходные данные сканера.

2. Когда сканер выделяет лексему "ошибочный тип"?

3. Зачем используется лексическая единица "конец исходного модуля"?

4. Какие незначащие элементы пропускает сканер?

5. Какие действия выполняет сканер? Сколько лексических единиц выделяет сканер за одно обращение к нему?

6. Какие типы ошибок может регистрировать сканер?

7. Как лучше организовать выделение арифметических констант — со знаком или без знака? Почему?

8. Какие дополнительные действия требуются, чтобы программа выдачи ошибок могла указать место ошибки в транслируемой программе?

9. Когда можно не использовать оператор *goto* в программе сканера? При каких условиях оператор *goto* имеет смысл оставить в программе сканера?

10. Что называется лексемой? Как описать лексему в программе компилятора?

11. Чем отличается программа, моделирующая работу заключительного и начального состояния?

12. Как запрограммировать состояние конечного автомата?

13. Какие действия начинают работу сканера?

14. Чем отличается программа, моделирующая работу заключительного и незаключительного состояния?

15. Что является глобальными данными для сканера?

16. Что представляет собой таблица лексических единиц? Зачем она нужна?

17. Какие способы программирования конечного автоматы вы знаете? Чем отличаются эти способы?

18. Как сканер осуществляет выделение ключевых слов?

19. Как построить общий проект компилятора?

20. Как проводить отладку сканера?

4.7 Тесты для самоконтроля к разделу 2

1. Какой из указанных на рисунке 4.8 автоматов правильно распознает десятичные вещественные константы с фиксированной точной языка C++?

Правильный ответ: 2.

2. Какое из следующих утверждений является правильным:

а) явный способ программирования сканера более экономичен с точки зрения времени работы программы;

б) как явный, так и неявный способ программирования сканера дает в конечном итоге программы с одинаковыми временными характеристиками; эти программы различаются только по объемам используемой памяти;

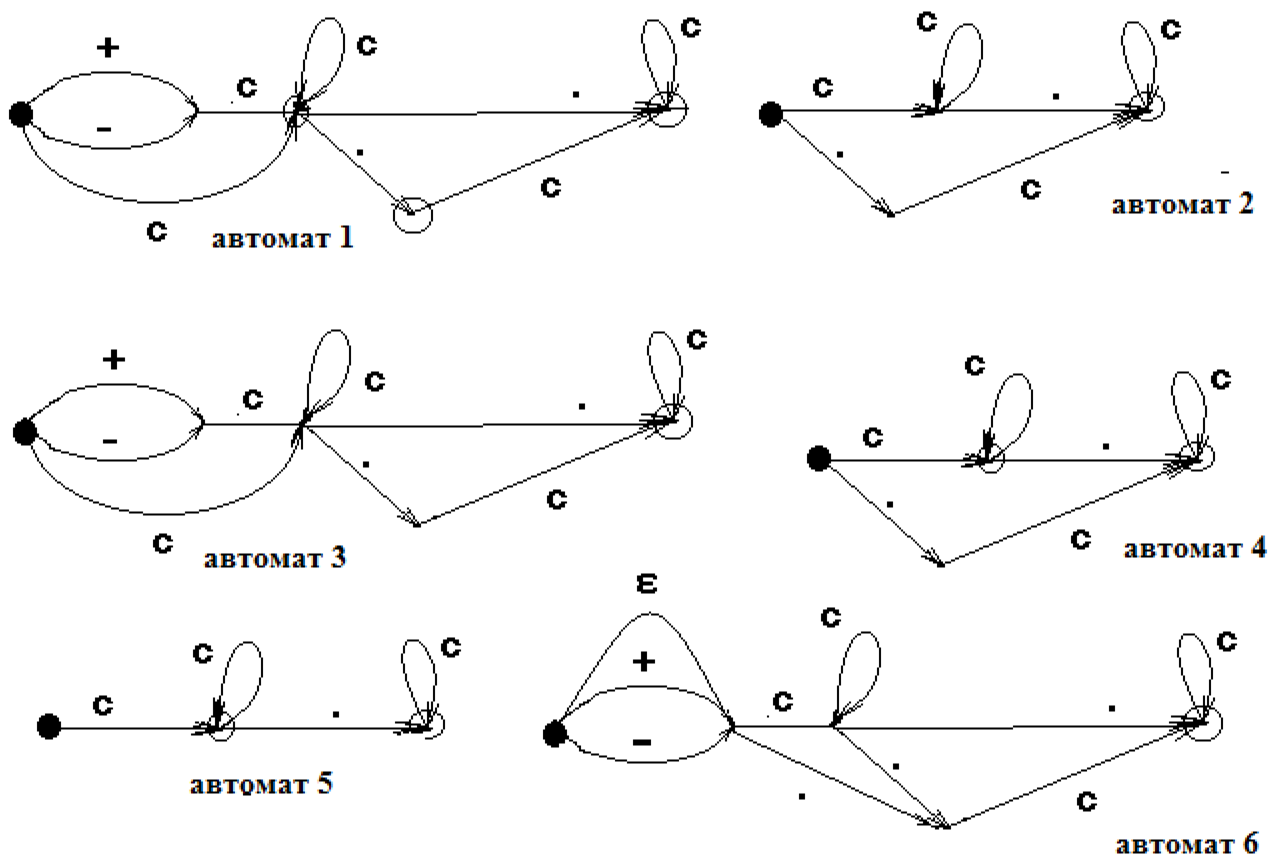


Рис. 4.8: Автоматы к тесту 1

в) неявный способ программирования сканера более экономичен с точки зрения времени работы программы;

г) явный и неявный способы программирования сканера дают в конечном итоге программы с одинаковыми временными характеристиками и объемами используемой памяти; эти программы различаются только принципом программирования.

Правильный ответ: в.

3. Какие из перечисленных ниже действий необходимо выполнить в начале функции сканера?

1) Ввод текста исходного модуля.

2) Очистка поля, предназначенного для хранения изображения лексемы.

3) Пропуск окончания слишком длинных лексем.

4) Пропуск пробелов, знаков табуляции и т.п.

5) Выделение ключевого слова, которое начинает очередной оператор.

Варианты ответов:

а) 1 и 5;

б) 2 и 5;

в) 1 и 4;

г) 2, 3 и 4;

д) 2 и 4.

Правильный ответ: д.

4. Что такое транслитератор?

Варианты ответов:

а) это автомат, который распознает лексические единицы языка;

б) это автомат, предназначенный для распознавания одной единственной лексической единицы языка;

в) это автомат, который переводит каждый ASCII–символ в его тип;

г) это программа, которая игнорирует незначащие символы языка;

д) это программа, которая распознает ключевые слова языка программирования;

е) это программа для выделения специальных знаков языка программирования;

ж) это программа, которая пропускает комментарии языка программирования.

Правильный ответ: в.

5. Какое поле или несколько полей из ниже перечисленных должны быть определены в таблице лексем?

Варианты ответов:

а) действия при неправильной структуре лексемы;

б) длина лексемы;

в) символ начала лексемы;

г) символ–ограничитель лексемы;

д) состояния конечного автомата, распознающего лексему.

Правильный ответ: г.

4.8 Упражнения к разделу

4.8.1 Задание

Цель данного задания – написать программу сканера, реализующего лексический уровень языка, КС–грамматику которого Вы построили в упражнении к главе 1. Выполнение задания организовать следующим образом.

1. Построить таблицу лексем языка программирования для Вашего задания на основе КС–грамматики, которую Вы построили, выполняя упражнения к разделу 1. В таблице отметить все признаки окончания лексем. Назначить обозначения типов для каждой лексемы. Ввести дополнительные обозначения для специальных лексем — окончания исходного модуля и ошибочного типа лексемы.

2. Для каждого типа лексемы из построенной Вами таблицы построить конечный автомат, допускающий соответствующие лексемы.

3. Построить общий конечный автомат лексического уровня Вашего задания. Выполнить детерминизацию построенного автомата.

4. Разметить все заключительные состояния построенного Вами конечного автомата, указав в них соответствующие типы выделяемых лексических единиц.

5. Построить список игнорируемых символов (например, знаков пробелов, табуляции и т.п.). Построить конечный автомат для комментариев, которые допускаются в языке программирования Вашего задания.

6. Определить программно типы данных, соответствующие исходному модулю и лексической единице. Определить заголовок программы сканера.

7. Написать программу сканера в соответствии с конечным автоматом, который Вы построили. Предусмотреть выдачу сообщения о лексических ошибках. Предусмотреть ограничение длины выделяемой лексемы, если Ваш конечный автомат может допускать лексемы бесконечной длины.

8. Написать процедуру выдачи сообщения об ошибке.

9. Написать главную программу, в которой необходимо предусмотреть:
- чтение файла с исходным модулем и вставку символа — маркера конца;
- вызов сканера до тех пор, пока не достигнут конец исходного модуля;
- вывод отсканированной лексемы и ее типа.

10. Построить проект системы программирования Microsoft VS C++, содержащий
- главный модуль;
- модуль сканера.

Все описания типов определить в отдельном модуле описаний $defs.hpp$.

11. Отладить программу на правильных и неправильных лексических конструкциях в исходном модуле.

4.8.2 Пример выполнения задания

Выполняя упражнения к главе 1, мы построили КС–грамматику упрощенного варианта языка Java-Script. Построим теперь таблицу лексем в соответствии с теми понятиями, которые мы вынесли на лексический уровень (см. рис 4.11). Построенный по обычным правилам конечный автомат лексического уровня представлен на рис. 4.9. Отмеченные на схеме заключительные состояния этого автомата соответствуют таблице лексических единиц.

Построим список игнорируемых символов:
– знаки пробелов,
– знак табуляции,
– знак перевода строки,
– комментарии, представляющие собой конструкцию от знаков "//" до конца строки.

Построим конечный автомат для комментариев, которые допускаются в языке программирования (см. рис 4.10). Этот конечный автомат не встраивается в общий автомат, соответствующий лексике языка программирования по тем причинам, что комментарии не являются значимыми терминальными символами на синтаксическом уровне и сканером пропускаются. Формальное изображение комментариев в виде конечного автомата требуется затем, чтобы в процессе программирования пропуска символов не допустить ошибки.

Все подготовительные операции выполнены, теперь можно приступать к программированию сканера. В проект включим четыре программных модуля, как это было рекомендовано в 2.5. В программе сканера будем минимизировать число лишних переходов с помощью оператора *goto*. Если какой–либо фрагмент конечного автомата представляет собой линейную последовательность действий, то проще и нагляднее будет и последовательное программирование соответствующих элементов в виде последовательности операторов. Переходы необходимо ставить при нелинейной структуре конечного автомата, как, например, это наблюдается для автомата, описывающего начало числовых констант.

```
//**********************************************************
// модуль defs.hpp ---  общие типы данных и макроопределения
//**********************************************************
#ifndef __DEFS
#define __DEFS
#include <stdio.h>
```

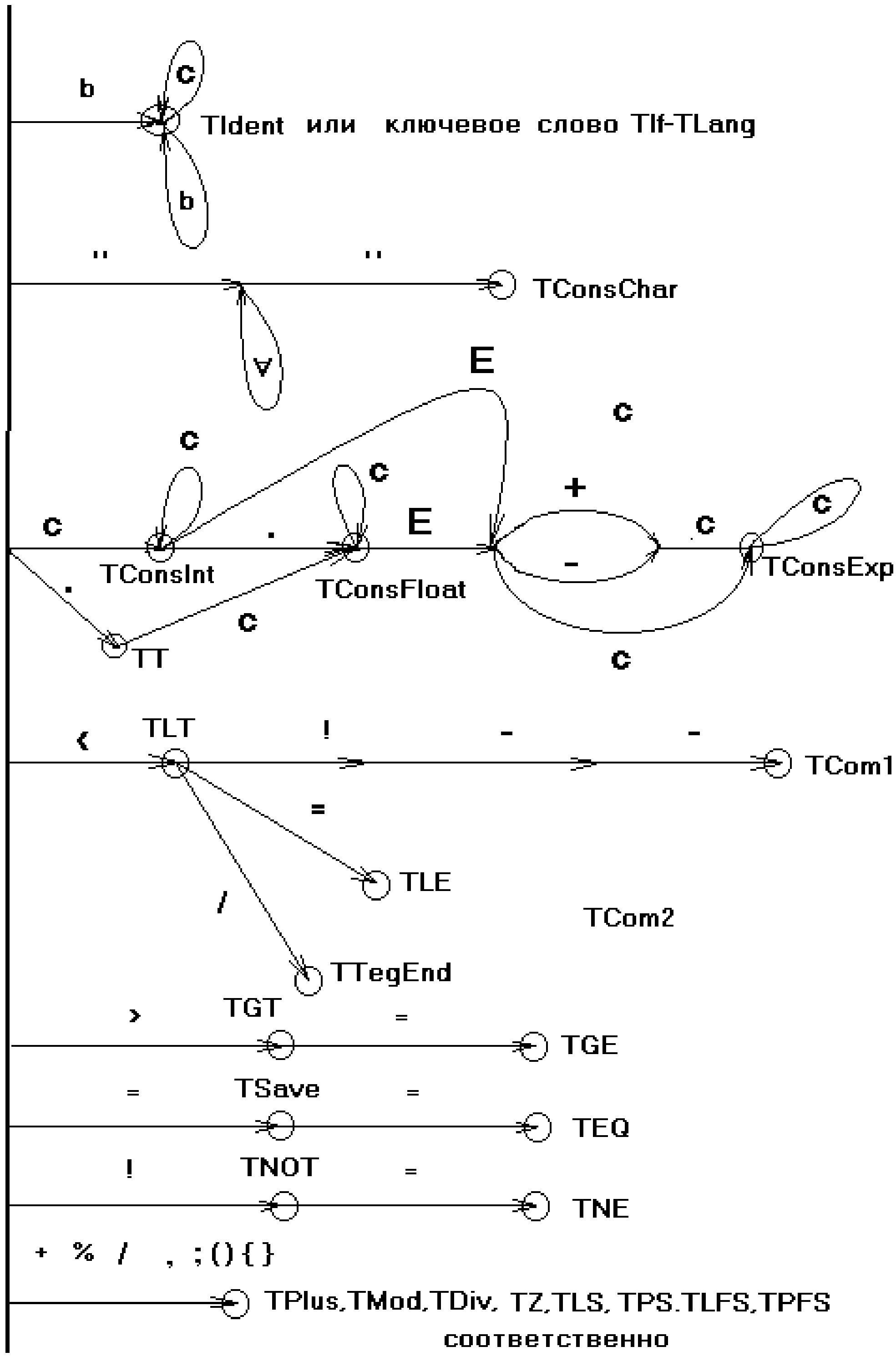


Рис. 4.9: Конечный автомат, представляющий лексику языка к упражнению 2.

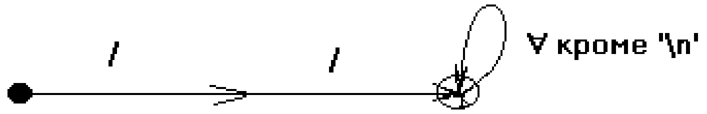


Рис. 4.10: Конечный автомат, представляющий комментарии к упражнению 2.

```
#include <stdlib.h>
#include <string.h>

#define MAX_TEXT 10000  // максимальная длина текста
#define MAX_LEX    100  // максимальная длина лексемы
#define MAX_KEYW     8  // число ключевых слов

typedef char TypeLex[MAX_LEX];
typedef char TypeMod[MAX_TEXT];

// ключевые слова
#define   TIf         1
#define   TFunct      3
#define   TVar        4
#define   TElse       5
#define   TScript     6
#define   TJava       7
#define   TLang       8
#define   TFor        9
// идентификаторы и константы
#define   TIdent     20
#define   TConsChar  30
#define   TConsInt   31
#define   TConsFloat 32
#define   TConsExp   33
// специальные знаки
#define   TToch      40
#define   TZpt       41
#define   TTZpt      42
#define   TLS        43
#define   TPS        44
#define   TFLS       45
#define   TFPS       46
// знаки операций
#define   TLT        50
#define   TLE        51
#define   TGT        52
#define   TGE        53
#define   TEQ        54
#define   TNEQ       55
#define   TPlus      56
#define   TMinus     57
#define   TMult      58
#define   TDiv       59
#define   TMod       60
#define   TSave      61
// знаки  тегов
#define   TTegEnd    70
#define   TCom1      71
#define   TCom2      72
```

```
// конец исходного модуля
#define   TEnd      100
// ошибочный символ
#define   TErr      200
#endif

//**********************************************************
//          модуль Scaner.hpp  ---  класс сканера
//**********************************************************
#ifndef __SCANER
#define __SCANER
#include "defs.hpp"

class TScaner{
private:
   TypeMod t;        // исходный текст
   int uk;           // указатель текущей позиции в исходном тексте


public:
   void PutUK  (int i);
   int  GetUK  (void);
   void PrintError(char *, char *);
   int  Scaner (TypeLex l);
   void GetData(char *);
   TScaner(char *);
   ~TScaner() {}
};
#endif



//**********************************************************
//модуль Scaner.cpp --- реализация методов класса сканера
//**********************************************************
#include <string.h>
#include "defs.hpp"
#include "Scaner.hpp"

TScaner:: TScaner(char * FileName) {
GetData(FileName);
PutUK(0);
}

TypeLex Keyword[MAX_KEYW]={ "if",          "for",      "function",
                            "var",         "else",     "script",
                            "javascript",  "language"
                          };
int  IndexKeyword[MAX_KEYW]={TIf,          TFor,      TFunct,
                             TVar,         TElse,     TScript,
```

```
                             TJava,        TLang
                          };

void   TScaner::PutUK(int i){uk=i;}     //восстановить указатель

int   TScaner::GetUK(void){return uk;}  // запомнить указатель

void   TScaner::PrintError(char * err, char * a)  {
     // выдать сообщение об ошибке
if (a[0]=='\0')
    printf("Ошибка :  %s %s\n",err,a);
    else
    printf("Ошибка :  %s. Неверный символ %s\n",err,a);
exit(0);
}

int   TScaner::Scaner(TypeLex l)  {
int i;                 // текущая длина лексемы
for (i=0;i<MAX_LEX;i++) l[i]=0;        //очистили поле лексемы
i=0;                  // лексема заполняется с позиции i
start:                 // все игнорируемые элементы:
while((t[uk]==' ') || (t[uk]=='\n') || (t[uk]=='\t')) uk++;
       // пропуск незначащих элементов
if ( (t[uk]=='/') && (t[uk+1]=='/') )
  { // начался комментарий, надо пропустить текст до '\n'
  uk=uk+2;
  while  ( (t[uk]!='\n')&&(t[uk]!='#'))  uk++;
  goto start;
  }
if (t[uk]=='\0')   {l[0]='#'; return TEnd;}
if ( (t[uk]<='9')&&(t[uk]>='0'))
       {
       l[i++]=t[uk++];
       while ( (t[uk]<='9')&&(t[uk]>='0'))
     if (i<MAX_LEX-1) l[i++]=t[uk++]; else uk++;
       if (t[uk]=='.') {l[i++]=t[uk++]; goto N1;}
       if( (t[uk]=='E')||(t[uk]=='e') ) { l[i++]=t[uk++]; goto N2; }
       return TConsInt;
       }
else  if ( (t[uk]>='a')&&(t[uk]<='z') ||
       (t[uk]>='A')&&(t[uk]<='Z') )
       {  //  начинается идентификатор
       l[i++]=t[uk++];
       while ( (t[uk]<='9')&&(t[uk]>='0') ||
       (t[uk]>='a')&&(t[uk]<='z') ||
       (t[uk]>='A')&&(t[uk]<='Z') )
     if (i<MAX_LEX-1) l[i++]=t[uk++]; else uk++;
   // длинный идентификатор обрезали
       int j;      // проверка на ключевое слово:
       for (j=0; j<MAX_KEYW; j++)
```

```
   if (strcmp(l,Keyword[j])==0) return IndexKeyword[j];
       return TIdent;
       }
else  if (t[uk]=='.')
       {
       l[i++]=t[uk++];
       if( (t[uk]<='9')&& (t[uk]>='0') ) { l[i++]=t[uk++]; goto N1; }
       return TToch;
       }
else  if (t[uk]=='\"')
       { uk++;  //  не будем включать кавычки в константу
       while( (t[uk]!='\"') && (t[uk]!='#') && (t[uk]!='\n'))
     {
     if (i<MAX_LEX-1) l[i++]=t[uk++]; else uk++;
     }
       if (t[uk]!='\"')
   {
   PrintError("Неверная символьная константа",l);
   return TErr;
   }
       uk++; // закрывающая кавычка
       return TConsChar; }
else  if (t[uk]==',')
       { l[i++]=t[uk++]; return TZpt; }
else  if (t[uk]==';')
       { l[i++]=t[uk++]; return TTZpt; }
else  if (t[uk]=='(')
       { l[i++]=t[uk++]; return TLS; }
else  if (t[uk]==')')
       { l[i++]=t[uk++]; return TPS; }
else  if (t[uk]=='{')
       { l[i++]=t[uk++]; return TFLS; }
else  if (t[uk]=='}')
       { l[i++]=t[uk++]; return TFPS; }
else  if (t[uk]=='+')
       { l[i++]=t[uk++]; return TPlus; }
else  if (t[uk]=='-')
       {
       l[i++]=t[uk++];
       if ((t[uk]=='-') && (t[uk+1]=='>') )
             {
             l[i++]=t[uk++]; l[i++]=t[uk++];
             return TCom2;
             }
       return TMinus;
       }
else  if (t[uk]=='/')
       { l[i++]=t[uk++]; return TDiv; }
else  if (t[uk]=='%')
       { l[i++]=t[uk++]; return TMod; }
```

```
else  if (t[uk]=='*')
       { l[i++]=t[uk++]; return TMult; }
else  if (t[uk]=='<')
       {
       l[i++]=t[uk++];
       if (t[uk]=='=') { l[i++]=t[uk++]; return TLE; }
       if (t[uk]=='/') { l[i++]=t[uk++]; return TTegEnd; }
       if ((t[uk]=='!') && (t[uk+1]=='-') && (t[uk+2]=='-') )
             {
             l[i++]=t[uk++]; l[i++]=t[uk++]; l[i++]=t[uk++];
             return TCom1;
             }
       return TLT;
       }
else  if (t[uk]=='>')
       {
       l[i++]=t[uk++];
       if (t[uk]=='=') { l[i++]=t[uk++]; return TGE; }
          else return TGT;
       }
else  if (t[uk]=='!')
       {
       l[i++]=t[uk++];
       if (t[uk]=='=') { l[i++]=t[uk++]; return TNEQ; }
          else { PrintError("Неверный символ",l); // ошибка
               return TErr;
               }
          }
else  if (t[uk]=='=')
       {
       l[i++]=t[uk++];
       if (t[uk]=='=') { l[i++]=t[uk++]; return TEQ; }
          else return TSave;
       }
else { PrintError("Неверный символ",l); // ошибка
     uk++;
     return TErr;
     }

N1:          // продолжение числовой константы после точки
while ( (t[uk]<='9')&&(t[uk]>='0'))
       if (i<MAX_LEX-1) l[i++]=t[uk++]; else uk++;
if ((t[uk]=='e')||(t[uk]=='E'))   { l[i++]=t[uk++]; goto N2; }
return TConsFloat;

N2:          // продолжение числовой константы после "E"
if ( (t[uk]=='+') || (t[uk]=='-') )
       {
       l[i++]=t[uk++];
       if ((t[uk]<='9')&&(t[uk]>='0'))
```

```
          {
          if (i<MAX_LEX-1) l[i++]=t[uk++]; else uk++;
          goto N3;
          }
          else
              {
              PrintError("Неверная  константа",l); // ошибка
              return TErr;
              }
      }

N3:          // продолжение порядка числовой константы
while ((t[uk]<='9')&&(t[uk]>='0'))
   {
   if (i<MAX_LEX-1) l[i++]=t[uk++]; else uk++;
   }
return TConsExp;
} // конец Scaner

void   TScaner::GetData(char * FileName)  {
     // ввод файла FileName, который содержит текст исходного модуля
char aa;
FILE * in  = fopen(FileName,"r");
if (in==NULL) { PrintError("Отсутствует входной файл",""); exit(1); }
int i=0;
while(!feof(in))
    {
    fscanf(in,"%c",&aa);
    if (!feof(in)) t[i++]=aa;
    if (i>=MAX_TEXT-1)
       {
       PrintError("Слишком большой размер исходного модуля","");
       break;
       }
    }
t[i]='\0';   // приписываем знак '\0' в конец текста
fclose(in);
} // конец GetData()

//**************************************************************
//  Главная программа транслятора -  отладочный вариант,
//                    предназначенный для отладки сканера
//**************************************************************
#include <stdio.h>
#include <string.h>
#include "defs.hpp"
#include "Scaner.hpp"

int main(int argc, char * argv[])  {
TScaner  * sc ;
```

```
int type; TypeLex l;
if (argc<=1) sc = new TScaner("input.txt");// файл по умолчанию
        else sc = new TScaner(argv[1]);    // заданный файл
do {
   type=sc->Scaner(l);
   printf("%s  - тип  %d \n",l, type);
   } while(type!=TEnd);
return 0;
}
```

Обратите внимание на тот факт, что при программировании лексических конструкций, заключенных в парные скобочные символы, необходимо предусмотреть анализ на конец исходного модуля в том случае, когда второй парный символ отсутствует. Такими лексемами в нашем примере являются комментарии (нет знака "$\backslash n$") и строковая константа (нет знака закрывающейся кавычки). Будем считать, что строковая константа не может быть многострочной, тогда дополнительно, кроме знака конца исходного модуля необходимо проверять и на знак завершения строки.

Лексические единицы языка	тип лексемы	символ–ограничитель
ключевые слова		
if	TIf=1	не буква, не цифра
for	TFor=2	—”—-
function	TFunct=3	—”—-
var	TVar=4	—”—-
else	TElse=5	—”—-
script	TScript=6	—”—-
javascript	TJava=7	—”—-
language	TLang=8	—”—-
идентификаторы	TIdent=20	не буква, не цифра
строковые константы	TConsChar =30	любой символ
константы целые	TConsInt =31	не цифра, не точка, не E
константы вещественные с точкой	TConsFloat=32	не цифра, не E
константы в экспоненциальной форме	TConsExp =33	не цифра
специальные знаки		
.	TToch=40	любой символ
,	TZpt=41	—”—-
;	TTZpt=42	—”—-
(	TLS=43	—”—-
)	TPS=44	—”—-
{	TFLS=45	—”—-
}	TFPS=46	—”—-
знаки операций		
<	TLT=50	не знак =, /, !
<=	TLE=51	любой символ
>	TGT=52	не знак =
>=	TGE=53	любой символ
==	TEQ=54	—”—-
! =	TNEQ=55	—”—-
+	TPlus=56	—”—-
−	TMinus=57	не знак -
*	TMult=58	любой символ
/	TDiv=59	—”—-
%	TMod=60	—”—-
=	TSave=61	не знак =
знаки тегов		
< /	TTegEnd=70	любой символ
<! − −	TCom1=71	—”—-
−− >	TCom2=72	—”—-
конец исходного модуля	TEnd=100	
ошибочный символ	TErr=200	

Рис. 4.11: Таблица лексем из упражнения к главе 2

Глава 5

МЕТОД РЕКУРСИВНОГО СПУСКА

Известная теорема о взаимно–однозначном соответствии КС–языков и МП–автоматов означает принципиальную необходимость использования магазина при анализе КС–языков. Рассмотрим другой, но тесно связанный с магазинными методами, метод рекурсивного спуска, который построен на замене механизма МП–автомата механизмом рекурсивного вызова процедур. Основная идея метода рекурсивного спуска состоит в том, что каждому нетерминалу грамматики соответствует процедура, которая распознает цепочку, порождаемую этим нетерминалом. Эти процедуры вызывают друг друга, когда это требуется.

Метод прост в реализации, т.к. основан на использовании рекурсивного вызова функций. Программисту не нужно создавать магазин и обеспечивать работу с ним. В ходе выполнения программы автоматически используется стек возвратов. Но следует учесть, что на самом деле система реализации вызовов существенно более сложна, чем требуется для реализации автомата с магазинной памятью (МП–автомата). Это означает, что прямая реализация магазина будет эффективнее по затратам времени и памяти, чем рекурсивный спуск. Разница в эффективности будет тем больше, чем более сложной является грамматика. Кроме того, при наличии сложной грамматики система рекурсивных процедур становится сложной в отладке. Таким образом, *метод рекурсивного спуска эффективен для реализации небольших языков программирования.*

5.1 Построение синтаксических диаграмм

Определение 5.1. Синтаксическая диаграмма (СД) — это ориентированный граф, построенный по правилам КС–грамматики $G = (V_T, V_N, P, S)$ и соответствующий отдельному нетерминальному символу множества V_N. Вершины этого графа представляют собой терминальные и нетерминальные символы КС–грамматики G, а дуги определяют связь символов в совокупности правил P для данного нетерминала.

Рассмотрим условные обозначения. Вершинами графа синтаксической диаграммы могут быть

а) терминальные символы $a \in V_T$; обозначаются скругленными блоками, например, *идентификатор*, ключевое слово *begin*, знак :=, и т.д;

б) нетерминальные символы; обозначаются прямоугольными блоками, например, конструкции *программа*, *выражение*, *составной оператор*.

Рис. 5.1: Терминальные элементы синтаксических диаграмм

Рис. 5.2: Нетерминальные элементы синтаксических диаграмм

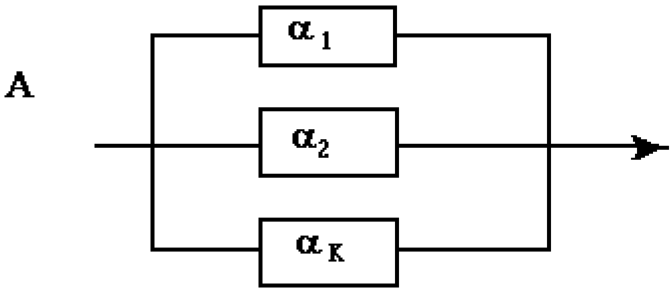
Рассмотрим способ построения диаграмм для КС–грамматик. Для этого надо воспользоваться следующими правилами.

1) Каждому нетерминалу ставится в соответствие синтаксическая диаграмма.

2) Если для нетерминального символа имеется несколько правил КС–грамматики

$$A \to \alpha_1|\alpha_2|...|\alpha_k,\ \ \alpha_i \in (V_T \cup V_N)^*,$$

то соответствующая диаграмма имеет вид:



3) Если в правиле $A \to \phi_i$ цепочка ϕ_i состоит из последовательности нетерминальных или терминальных символов a_1, a_2, ... , a_k, то ветвь синтаксической диаграммы имеет вид, представленный на рис. 5.3.

Рассмотрим пример описания оператора if, который может содержать вложенные операторы if, операторы присваивания и составные операторы. Такой язык можно описать, например, следующей КС–грамматикой:

$$\begin{array}{ll} G: & S \to if(V)O \mid if(V)O\ else\ O \\ & O \to S \mid H \mid D \\ & H \to a = V; \\ & D \to \{P\} \\ & P \to PO \mid \varepsilon \\ & V \to +A \mid -A \mid V+A \mid V-A \mid A \\ & A \to A*E \mid A/E \mid E \\ & E \to a \mid c \mid (V) \end{array}$$

Здесь нетерминалы имеют следующий смысл: S — сам оператор if, O — любой из вложенных операторов, H — оператор присваивания, D — составной оператор, P — последовательность операторов (в том числе и пустая). Нетерминальные символы V, A, E используются для определения выражения с операциями двух уровней приоритета.

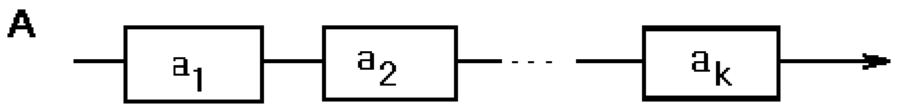


Рис. 5.3: Диаграмма для правила $A \to a_1 a_2 ... a_k$

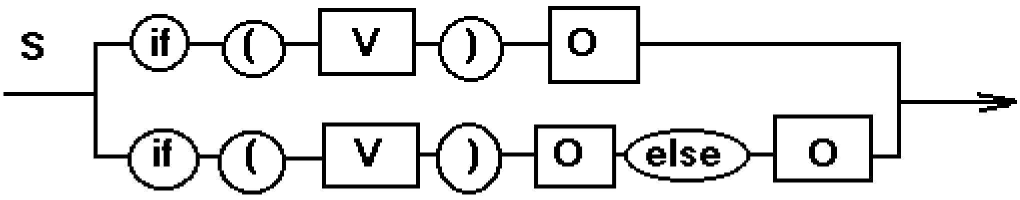


Рис. 5.4: Синтаксическая диаграмма для $S \to if(V)\ O \mid if(V)\ O\ else\ O$

Для нетерминала S в грамматике имеется два правила, поэтому синтаксическая диаграмма для нетерминала S имеет две параллельные ветви. В каждой из них терминалам (круглым скобкам, ключевым словам if и $else$) и нетерминальным символам V и O соответствует один блок — скругленный или прямоугольный соответственно. Напротив, для нетерминала H имеется только одно правило грамматики и, следовательно, синтаксическая диаграмма H содержит только одну ветвь, состоящую из четырех элементов — трех терминальных блоков и одного нетерминального V.

Все построенные по данным правилам синтаксические диаграммы представлены на рис. 5.1 - 5.11.

Построенные диаграммы, как правило, сложны, недетерминированы и рекурсивны. Надо по возможности преобразовать такие диаграммы с целью их упрощения и детерминизации. Очевидно, что любой бесконечный язык может быть задан только на основе рекурсивных языковых конструкций. Тогда в построенных по изложенным выше правилам синтаксических диаграммах обязательно наличие рекурсии. Рекурсия может быть левой, правой и центральной (в том числе и множественной). От центральной рекурсии в синтаксических диаграммах избавиться невозможно, однако, левая и правая рекурсии вполне устранимы.

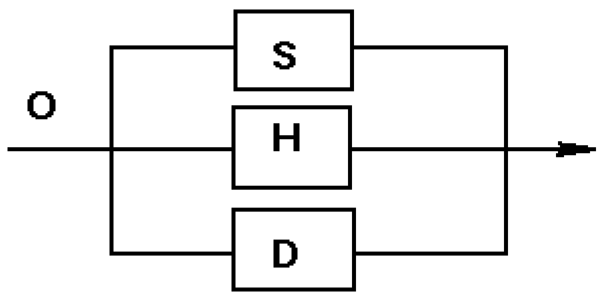


Рис. 5.5: Синтаксическая диаграмма для $O \to S|H|D$

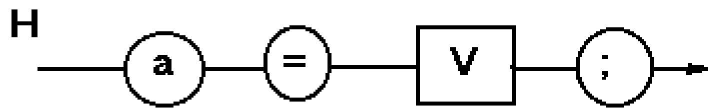


Рис. 5.6: Синтаксическая диаграмма для $H \to a = V;$

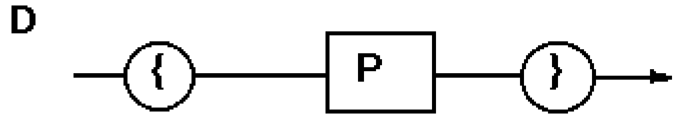


Рис. 5.7: Синтаксическая диаграмма для $D \to \{P\}$

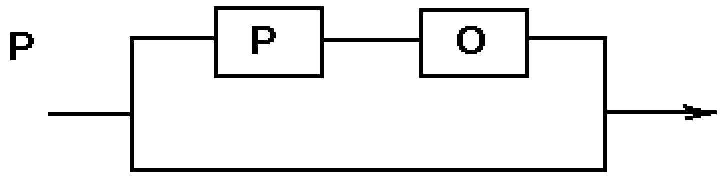


Рис. 5.8: Синтаксическая диаграмма для $P \to PO|\varepsilon$

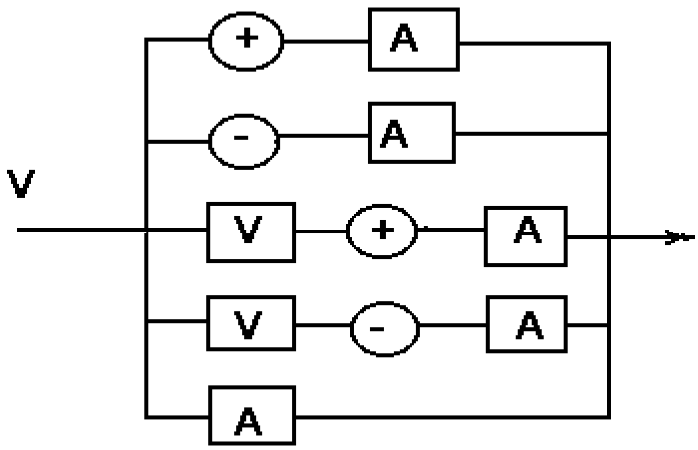


Рис. 5.9: Синтаксическая диаграмма для $V \to +A| - A|V + A|V - A|A$

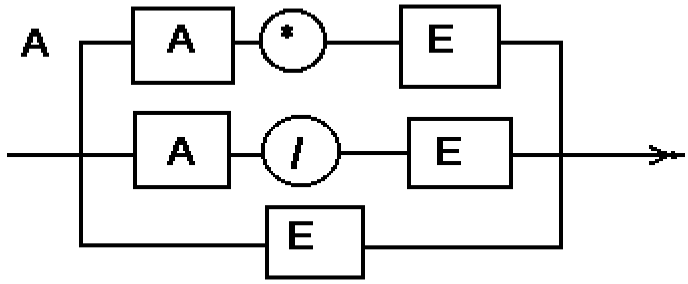


Рис. 5.10: Синтаксическая диаграмма для $A \to A * E|A/E|E$

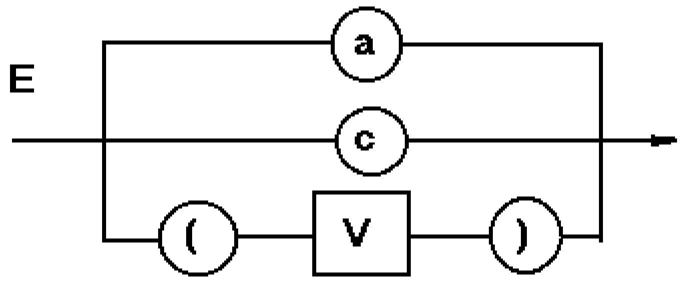


Рис. 5.11: Синтаксическая диаграмма для $E \to a|c|(V)$

5.2 Преобразование синтаксических диаграмм

5.2.1 Вынесение левых и правых множителей

Очень часто две или более ветвей синтаксической диаграммы имеют одинаковые начала и окончания, связанные с наличием общих префиксов или суффиксов в правилах КС–грамматики:

$$A \to \beta\phi_1\gamma|\beta\phi_2\gamma|...|\beta\phi_k\gamma, \beta, \gamma, \phi_i \in (V_T \cup V_N)^*.$$

Очевидно, что общие префиксы и суффиксы можно вынести, заменив их на нетерминальные символы:

$$\begin{array}{ll} A & \to \beta B\gamma \\ B & \to \phi_1|\phi_2|...|\phi_k \end{array}$$

Такое эквивалентное преобразование грамматик означает возможность вынесения общих префиксов и суффиксов на уровне синтаксических диаграмм. Если две или более ветвей синтаксической диаграммы имеют одинаковые начала или одинаковые окончания, их можно вынести и создать одну общую ветвь (см. рис. 5.12).

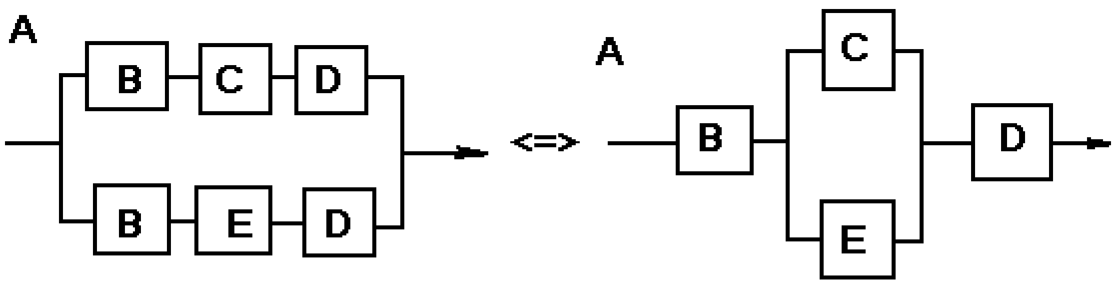


Рис. 5.12: Правило вынесения общих множителей в синтаксической диаграмме

Например, в диаграмме S, описывающей оператор if, следует вынести общее начало $if(V)O$ (см. рис. 5.4). Полученная диаграмма представлена на рис. 5.13.

Здесь следует отметить, что выносить общие множители в леворекурсивных или праворекурсивных диаграммах можно только с учетом их дальнейшего преобразования. Забегая немного вперед, можно сказать, что выносить общие множители в таких диаграммах следует во всех рекурсивных и во всех нерекурсивных ветвях раздельно. Применяя рассмотренные преобразования к диаграммам из нашего примера, получим диаграммы для нетерминалов V и A, представленные на рис. 5.14.

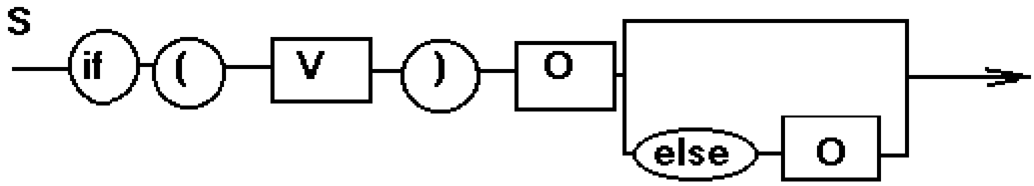


Рис. 5.13: Диаграмма S после вынесения общих множителей

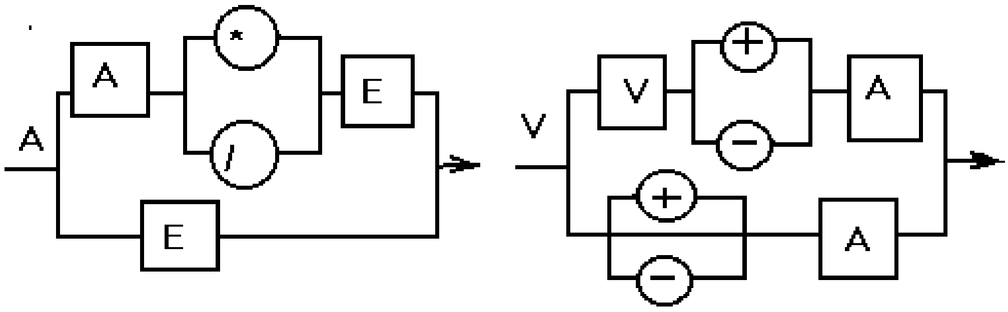


Рис. 5.14: Вынесение общих множителей в V и A

5.2.2 Удаление левой и правой рекурсии

Пусть есть леворекурсивная диаграмма, в которой имеется леворекурсивная и простая ветвь без рекурсии (см рис. 5.15).

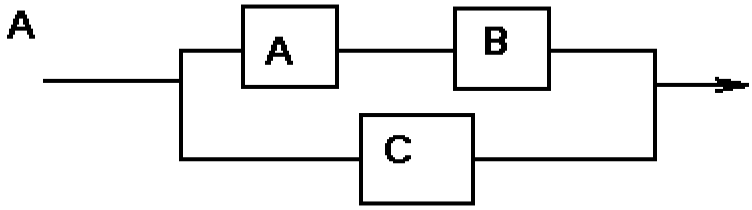


Рис. 5.15: Пример простой леворекурсивной синтаксической диаграммы

Здесь B и C — либо простые элементы, либо сложные конструкции. Рассмотрим вывод из нетерминала A:

$$A \Rightarrow AB \Rightarrow ABB \Rightarrow ... \Rightarrow CBB...B.$$

Следовательно, эквивалентные правила грамматики для нетерминала A имеют вид

$$\begin{array}{ll} A \rightarrow & CD \\ D \rightarrow & DB|\varepsilon. \end{array}$$

Тогда эквивалентная диаграмма имеют вид, представленный на рис. 5.16.

В языках программирования часто $B = kC$, например, список фактических параметров функции при ее вызове определяется правилами

$$<\text{ список }>\rightarrow<\text{ список }>,<\text{ выражение }>\ |\ <\text{ выражение }>.$$

В этом случае эквивалентная диаграмма, построенная по приведенным выше правилам, имеет вид, представленный на рис. 5.18.

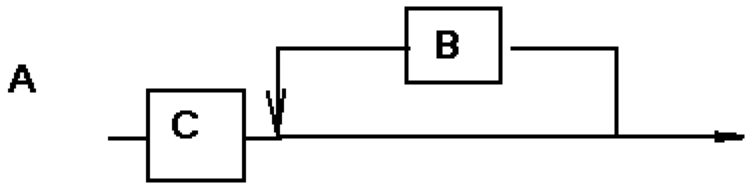


Рис. 5.16: Устранение левой рекурсии в $A \Rightarrow AB \Rightarrow ABB \Rightarrow ... \Rightarrow CBB...B$.

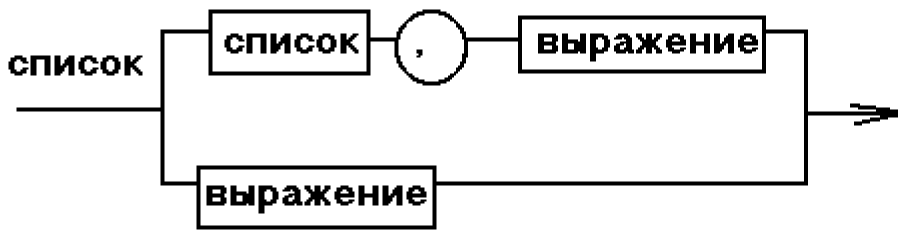


Рис. 5.17: Леворекурсивная диаграмма для нетерминала < список >

Однако, такое преобразование рекомендуется делать только тогда, когда синтаксический элемент k представляет собой один терминальный символ и не несет семантической нагрузки. Например, для рассмотренного выше понятия $<\{$ *список* $\}>$ в результате преобразования получим

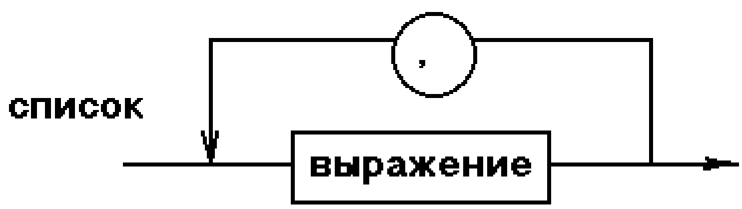


Рис. 5.18: Устранение левой рекурсии для нетерминала < список >

Попытка сократить конструкцию по рассмотренным правилам приведет к существенному усложнению программы при реализации контекстных условий языка программирования. Не рекомендуется также выполнять данную форму преобразования и в том случае, когда k представляет собой сложную конструкцию.

Рассмотрим теперь правую рекурсию. Строго говоря, правая рекурсия допустима при реализации синтаксического анализатора методом рекурсивного спуска, т.к. ее наличие приводит к принципиально реализуемой программе. Однако, из соображений эффективности построенной программы следует заменить рекурсивный вызов на нерекурсивную циклическую конструкцию. Процесс преобразования выполняется на основе преобразования правил КС–грамматики

$$A \to BA|C$$

к виду

$$\begin{array}{ll} A \to & DC \\ D \to & DB|\varepsilon. \end{array}$$

Следовательно, диаграмма рис. 5.19 преобразуется к виду рис. 5.20.

Примеры устранения левой рекурсии в диаграммах A и V после вынесения общих множителей представлены на рис. 5.21 и 5.22 соответственно.

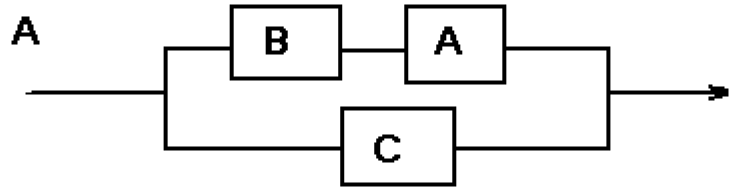


Рис. 5.19: Пример простой праворекурсивной синтаксической диаграммы

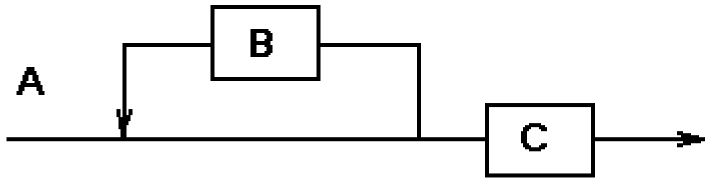


Рис. 5.20: Устранение правой рекурсии в правилах $A \rightarrow BA|C$

5.2.3 Подстановка диаграммы в диаграмму

Подстановка имеет смысл, когда имеются простые, состоящие из одного–двух блоков нерекурсивные диаграммы. При такой подстановке ссылки на подставляемые диаграммы должны отсутствовать в оставшихся диаграммах, в противном случае полученная диаграмма становится только сложнее.

В примере предшествующего параграфа можно заметить, что после устранения левой рекурсии диаграмму P можно подставить в диаграмму D (см. рис. 5.23). Вообще говоря, можно было бы выполнить и подстановки других диаграмм, например, диаграмм D в диаграмму O, или диаграммы A в диаграмму V, однако при такой подстановке существенно усложнилась бы структура полученной диаграммы. При выполнении подстановки следует, как правило, руководствоваться следующими принципами.

1) Всякая полученная в результате подстановки диаграмма должна быть простой; под простой диаграммой следует понимать такую, которая не содержит ветвлений и циклов в большом количестве (при этом даже двухуровневая вложенность нежелательна), не содержит длинных последовательностей блоков.

2) Следует учитывать, что с диаграммой, возможно, придется связывать семантические подпрограммы. Наличие в одной синтаксической диаграмме сложных и разных по семантическому наполнению ветвей также нежелательно.

3) Нет смысла оставлять линейные синтаксические диаграммы, состоящие из одного — двух блоков.

4) Главная цель — получить минимальное число максимально простых синтаксических диаграмм. Очевидно, что цель противоречива, поэтому главное — придерживаться "золотой середины".

5.3 Разметка ветвей синтаксических диаграмм

Для того, чтобы запрограммировать точки ветвления в синтаксической диаграмме, надо выяснить условия, по которым осуществляется переход на ту или иную

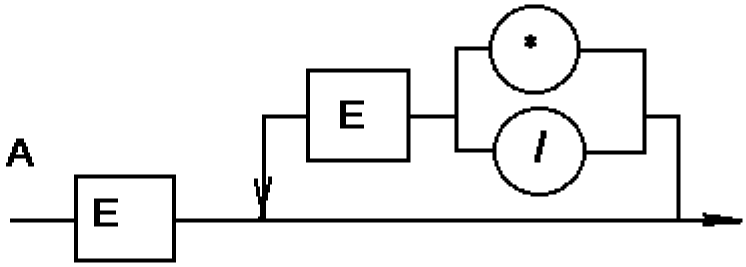


Рис. 5.21: Устранение левой рекурсии в A (см. рис. 5.10)

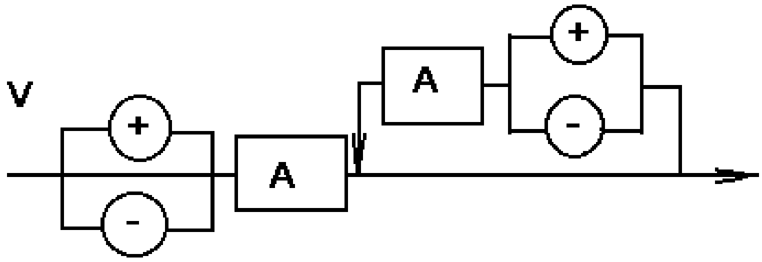


Рис. 5.22: Устранение левой рекурии в V (см. рис. 5.9)

ветвь. Для этого используются специальные функции, определенные в общем случае на множестве цепочек из терминальных и нетерминальных символов КС-грамматики:

$$first_k(Y), last_k(Y), follow_k(Y),$$

где k - натуральное число, а Y - цепочка. Эти функции определяют соответственно множество начал и концов цепочек, выводимых из Y, а также цепочки, следующие за Y в некоторой сентенциальной форме. Число k определяет длину таких цепочек. Таким образом, на неформальном уровне эти функции определяют множество терминальных цепочек длины не более k, которые являются

а) для $first_k(Y)$ — началом каждой цепочки, выводимой из Y;

б) для $follow_k(Y)$ — следующими за Y цепочками;

в) для $last_k(Y)$ — концом цепочек, выводимых из Y.

Если соответствующие множества $first_k(Y)$, $last_k(Y)$, $follow_k(Y)$ известны, то в точках ветвления синтаксической диаграммы разметка ветвей имеет вид, представленный на рис. 5.24.

Число k в функциях $first_k(Y)$, $follow_k(Y)$, $last_k(Y)$ определяет число сканируемых символов. Для повышения скорости работы транслятора диаграммы строятся так, чтобы выбрать $k = 1$. Если разметки ветвей не пересекаются, то диаграмма готова к программированию. Если имеются пересечения, необходимо либо увеличить k, либо перестроить диаграмму (и, возможно, грамматику). Следует, однако, отметить, что в некоторых случаях никакое преобразование грамматик не позволяет избавиться от неоднозначности при выборе переходов в точках ветвления. Это означает, что анализируемый язык является существенно неоднозначным и не может быть проанализирован безвозвратными методами. Для языков программирования грамматики должны быть однозначными, т.к. транслятор обрабатывает исходные модули больших размеров и любые возвратные методы синтаксического анализа приведут к неприемлемым временным характеристикам алгоритмов анализа. Фактически все языки программирования являются однозначными. Исключение, практически единственное для известных языков программирования, касается структуры условного

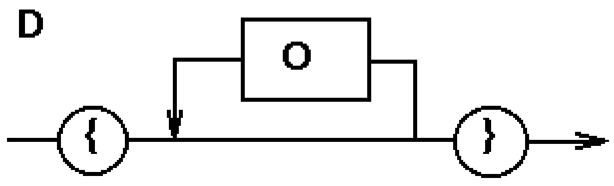


Рис. 5.23: Подстановка P (после устранения рекурсии) в D

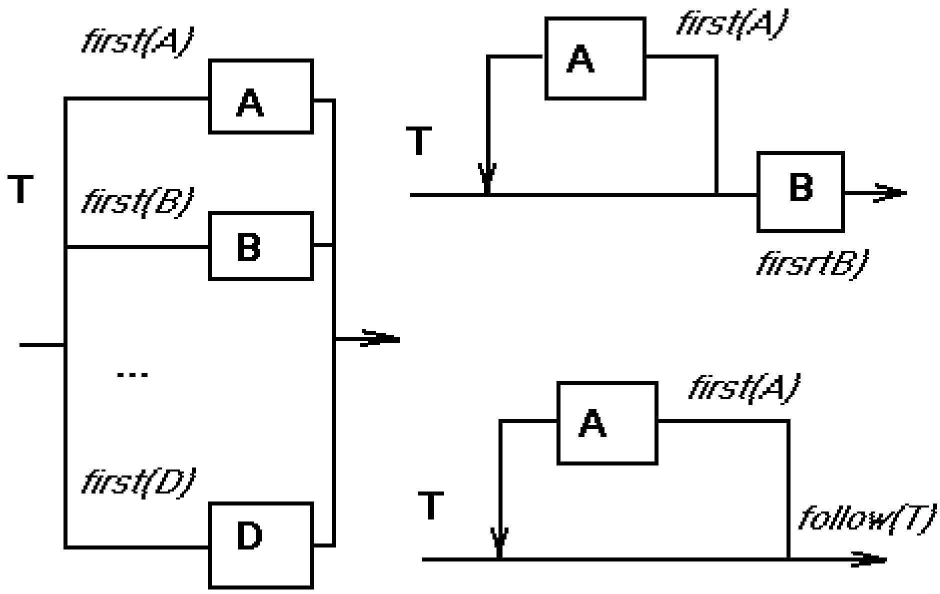


Рис. 5.24: Правила разметки точек ветвления в синтаксических диаграммах.

оператора if. Возможность записи условного оператора как в полной так и в усеченной форме

$$S \to \text{ if } V \text{ then } S \text{ else } S| \text{ if } V \text{ then } S|O,$$

приводит к неоднозначности разбора конструкции if V then if V then O else O:

а) if V then { if V then O else O },

б) if V then { if V then O } else O.

Причина неоднозначности — возможность сопоставления $else$ с двумя разными $then$. Вообще говоря, можно построить однозначную КС–грамматику для оператора if. Но такая грамматика получается весьма громоздкой и неудобной для анализа. Поэтому с учетом требований соответствия $else$ последнему из предшествующих $then$ при разметке ветвей синтаксических диаграмм искусственно удаляют $else$ из множества, помечающего пустую ветвь в диаграмме оператора if.

Отметим один самый важный факт, касающийся возможности построения безвозвратно работающего синтаксического анализатора методом рекурсивного спуска. *Только если разметка ветвей допускает однозначный выбор ветви при реализации переходов, применим метод рекурсивного спуска. Если однозначный выбор ветви невозможен, метод неприменим из–за необходимости реализации перебора вариантов, что приведет к экспоненциальной временной сложности алгоритмов грамматического разбора.*

5.4 Алгоритм построения функций $first$, $last$ и $follow$

Определение 5.2. В КС–грамматике $G = (V_T, V_N, P, S)$ функция $first_k(A)$ — это множество префиксов длины не более k всех терминальных цепочек, выводимых из A:

$$first_k(A) = \{x|x \in V_T^*; A \stackrel{*}{\Rightarrow} xy; (|x| = k \vee |x| < k\&|y| = 0)\}.$$

Учитывая рекурсивную природу КС–грамматик построим рекурсивный алгоритм определения функции $first$. Очевидно, что для любой цепочки $y = ab...c$ выполняется равенство

$$first_k(y) = first_k(first_k(a) \cdot first_k(b) \cdot ... \cdot first_k(c)).$$

Тогда достаточно применить алгоритм определения функции $first$ для символов, а при вычислении функции $first$ для цепочек достаточно построить произведение соответствующих множеств и для всех цепочек построенного множества вычислить все префиксы длины не более k. Для любого терминала a имеет место равенство $first_k(a) = a$. Для нетерминального символа A будем строить множество $first_k(A)$ последовательно:

$$\begin{array}{ll} first_k^0(A) & = \{x|A \to xy \in P; x \in V_T^*; (|x| = k \vee |x| < k\&|y| = 0)\} \\ first_k^{i+1}(A) & = first_k^i(A) \cup first_k^i(y), \text{ где} \\ & A \to y \in P, \\ & y = ab...c, \\ & first_k^i(y) = first_k(first_k^i(a) \cdot firstk_i(b) \cdot ... \cdot first_k^i(c)). \end{array}$$

Алгоритм заканчивается, когда очередное множество $first_k^{i+1}(A)$ совпадает с множеством $first_k^i(A)$.

Пример 3,1. Рассмотрим КС–грамматику

$$\begin{array}{ll} G: & S \to if(V)O \\ & O \to S|a = V; \\ & V \to V + A|V - A| + A| - A|A \\ & A \to A * T|A/T|T \\ & T \to a|c|(V) \end{array}$$

Построим функцию $first_1$ для нетерминалов этой грамматики. Процесс построения сведем в таблицу.

нетерминал	$i = 1$	$i = 2$	$i = 3$
S	if	if	if
O	a	if, a	if, a
V	$+$, $-$	$+$, $-$	$+$, $-$, a, c, (
A		a, c, (	a, c, (
T	a, c, (	a, c, (	a, c, (

Определение 5.3. В КС–грамматике $G = (V_T, V_N, P, S)$ функция $last_k(A)$ — это множество суффиксов длины не более k всех терминальных цепочек, выводимых из A:

$$last_k(A) = \{x|x \in V_T^*; A \stackrel{*}{\Rightarrow} yx; (|x| = k \vee |x| < k\&|y| = 0)\}.$$

Алгоритм построения функции $last$ является зеркальным отображением алгоритма построения функции $first$ и оставляется в качестве упражнения.

Рассмотрим теперь алгоритм построения функции $follow$. Будем считать, что исходный модуль дополнен справа k маркерами конца, т.е. цепочкой $\natural^k$. Тогда все цепочки, следующие за любым терминалом или нетерминалом в любой сентенциальной форме, будут длины не менее k. Такое дополнение справа цепочкой $\natural^k$ текста исходного модуля позволяет унифицировать процесс определения функции $follow$ в любой позиции сентенциальной формы.

Определение 5.4. В КС–грамматике $G = (V_T, V_N, P, S)$ функция $follow_k(A)$ — это множество терминальных цепочек, следующее за A в выводе из $S\natural^k$:

$$follow_k(A) = \{x|S\natural^k \overset{*}{\Rightarrow} yAB; B \overset{*}{\Rightarrow} xz; x \in (V_T \cup \{\natural\})^*; |x| = k\}.$$

Очевидно, что можно дать эквивалентное определение функции $follow$ с помощью функции $first$:

$$follow_k(A) = \{x|S\natural^k \overset{*}{\Rightarrow} yAz; x \in first_k(z)\}.$$

На основе данного определения можно построить следующий алгоритм вычисления функции $follow$ для нетерминальных символов.

1) Преобразуем грамматику к неукорачивающей форме. Это преобразование является временным преобразованием, неукорачивающая грамматика нужна только для построения функции $follow$.

2) Строим множество N всевозможных цепочек длины $k+1$, выводимых из цепочки $S\natural^k$. При этом цепочки, первый символ которых не является нетерминалом, в множество N можно не включать.

3) Строим множество

$$follow_k(A) = \{x|Ax \in N; x \in (V_T \cup \{\natural\})^*.$$

Пример 5.2. Построим функцию $follow_1$ для грамматики

$$\begin{array}{ll} G: & S->if(V)O \\ & O->S|a=V; \\ & V->V+A|V-A|+A|-A|A \\ & A->A*T|A/T|T \\ & T->a|c|(V) \end{array}$$

Грамматика неукорачивающая, поэтому преобразование не требуется. Строим множество

$$N = \{S\natural, O\natural, V;, V+, V-, A;, A+, A-, T;, T+, T-, V), A*, A/, A), T), T*, T/\}$$

Тогда значения функции $follow_1$ можно представить следующей таблицей:

нетерминал	S	O	V	A	T
$follow$	$\natural$	$\natural$	$)+-;$	$)+-*/;$	$)+-*/;$

Очень несложно автоматизировать процесс вычисления функции $first_1$ по правилам грамматики. Простейший вариант программы основан на представлении правил по строгому шаблону, так, что каждое правило записывается следующим образом:

1) одно правило занимает одну строчку исходного файла,

2) правило начинается символом нетерминала — большой латинской буквой,

3) после нетерминала указаны знаки $->$, отделяющие правую часть правила от левой части,

4) в качестве терминалов можно использовать любые символы ASCII, кроме больших латинских букв,

5) пустой символ обозначется знаком ♮.

Тогда программа вычисления функции $first_1$ может быть реализована как указано ниже:

```
#include <stdio.h>
#include <process.h>
#include <string.h>
#define MaxSym   26           //  максимальное число нетерминалов
#define MaxTerm 256           //  максимальное число символов
#define MaxStr 256            //  максимальное число правил

char left[MaxStr];            // левая часть правила
char right[MaxStr][70];       // правая часть правила
int kol;                      // число правил грамматики
short int f[MaxSym][MaxTerm]; // значение first_1
char Vn[MaxSym];              // список нетерминалов

void GetData(char * FileName)
// ввод правил грамматики (нетерминал - большая латинская буква)
// пустая цепочка представима символом #
{
char aa;
FILE * in  = fopen(FileName,"r");
if (in==NULL) { printf("Отсутствует входной файл",""); exit(1); }
int i;
kol=0;
while(!feof(in))
    {
    fscanf(in,"%c",&aa);
    if (!feof(in)) left[kol++]=aa;
    fscanf(in,"%c",&aa);     fscanf(in,"%c",&aa);
    // прочитан знак ->
    i=0; // длина правой части правила
    while(!feof(in))
       {
       fscanf(in,"%c",&aa);
       if (!feof(in) && (aa!='\n')) right[kol-1][i++]=aa;
   else break;
       }
    }
fclose(in);
}

void PrintFirst()
// вывести множество first_1
{
int i,j;
```

```
for (j=0; j<MaxSym; j++) if (Vn[j]!='\0')
  {
  printf(" first(%c) =      ",Vn[j]);
  for ( i=0; i<MaxTerm; i++) if ( f[j][i]!='\0')
      printf(" %c ",(char)i);
  printf(" \n");
  }
}

int Reset(int in, int ip,int m)
// обновление множества first для нетерминала in по правилу ip
// ip - номер правила,  in - номер нетерминала
// m - номер первого символа правила
{
int i, j, flag;
short ff[MaxTerm]; memset(ff,0,MaxTerm*sizeof(short));
   // ff - новое пополнение first_1
char n; n=right[ip][m];
if ( (n<='Z') && (n>='A') )
     memcpy(&ff, f[(int)n-(int)'A'],MaxTerm);
      // для нетерминала first добавляем к текущему множеству
    else ff[n]=1;  // для терминала first - сам этот терминал
flag=0;
for (i=0; i<MaxTerm; i++)
  {char a=f[in][i];
  if ( (i==(int)'#') && (ff[i]!=0) && (right[ip][m+1]!='\0') )
// не пустой символ или нет продолжения
flag=Reset(in,ip,m+1) || flag;
       else f[in][i]=f[in][i] | ff[i];
  if (f[in][i]!= a) flag=1;
  }
return flag;
}

void First(void)
// построить множество first_1
{
int flag,i,j,k;
for (i=0; i<MaxSym; i++)
  {
  Vn[i]='\0';
  for (j=0; j<MaxTerm; j++)f[i][j]='\0';
  }               // массивы очищены
for (i=0; i<kol; i++)
   {
   j=(int)left[i]-(int)'A';    Vn[j]=left[i];
   }              // список нетерминалов построен
flag=1;
while (flag)
   {
```

```
   flag=0;
   for ( i=0; i<kol; i++)        // i - номер правила
      {
      j=(int)left[i]-(int)'A';   // j - номер нетерминала
      k = Reset(j,i,0);
      flag=flag || k;
      }
   }
}

int main(int argc, char * argv[])
{
if (argc<=1) GetData("input.txt");// без параметра - файл по умолчанию
else GetData(argv[1]);    // ввести данные
First();
PrintFirst();
}
```

Пример описания в файле *input.txt* данных:

```
S->aSb
S-Ac
A->bbcA
A->#
```

При тех же соглашениях о представлении исходных данных алгоритм вычисления функции $folow_1$ может быть реализован следующим образом:

```
#include <stdio.h>
#include <process.h>
#include <string.h>
#define MaxSym     26          //  максимальное число нетерминалов
#define MaxTerm   256          //  максимальное число символов
#define MaxStr    256          //  максимальное число правил

char left[MaxStr];             // левая часть правила
char right[MaxStr][70];        // правая часть правила
int kol;                       // число правил грамматики
short int fw[MaxSym][MaxTerm]; // значение follow_1

void GetData(char * FileName)
// ввод правил грамматики (нетерминал - большая латинская буква)
// пустая цепочка представима символом #
{
char aa;
FILE * in  = fopen(FileName,"r");
if (in==NULL) { printf("Отсутствует входной файл",""); exit(1); }
int i;
kol=0;
while(!feof(in))
```

```
    {
    fscanf(in,"%c",&aa);
    if (!feof(in)) left[kol++]=aa;
    fscanf(in,"%c",&aa);     fscanf(in,"%c",&aa);
    // прочитан знак ->
    i=0; // длина правой части правила
    while(!feof(in))
       {
       fscanf(in,"%c",&aa);
       if (aa=='#')
  {
  printf("Грамматика должна быть неукорачивающей!\n");
  exit(0);
  }
       if (!feof(in) && (aa!='\n')) right[kol-1][i++]=aa;
   else break;
       }
    }
fclose(in);
}

void PrintPar()
// вывести множество всех пар символов
{
int i,j;
for (i=0; i<MaxSym; i++)
   {
   for (j=0; j<MaxTerm; j++)
       if (fw[i][j]!=0)
     printf(" %c%c ",i+'A',j);
   printf("\n");
  }
}

void PrintFollow()
// вывести множество follow_1  для всех символов грамматики
{
int i,j, flag;
for (i=0; i<MaxSym; i++)
   {flag=0;
   for (j=0; j<MaxTerm; j++)
       if ((fw[i][j]!=0) && ((j>'Z')|| (j<'A')) )
     {
     if (flag==0) printf(" follow(%c)=  ",i+'A');
     flag=1;
     printf(" %c, ",j);
     }
   if (flag) printf("\n");
  }
}
```

```
int SetNew(int iNet, int iSym)
// обновление множества пар N путем порождения из пары iNet-iSym
// iNet - нетерминал, iSym - второй символ
{
int i, j, k, flag, a, b, c;
flag=0;
for (i=0; i<kol; i++)            // по всем правилам грамматики
  if (left[i]==iNet)
      { k=0;
      do
 {
 a=right[i][k];
 if (right[i][k+1]!='\0')   b=right[i][k+1];
     else        b=iSym;
 if( (a<='Z') && (a>='A') )    // 1-ый символ - нетерминал
    {
    if (fw[a-(int)'A'][b]==0)
   { flag=1;  fw[a-(int)'A'][b]=1; printf("%c%c ",a,b);}
    if( (b<='Z') && (b>='A') ) // 2-ой символ - нетерминал
    for (j=0; j<kol; j++)      // по всем правилам грамматики
       if (left[j]==b)
       {
       c=right[j][0];
       if (fw[a-(int)'A'][c]==0)
                   {flag=1; fw[a-(int)'A'][c]=1; printf("%c%c ",a,c);}
       }
    }
 k++;
 } while(right[i][k]!='\0');
      }
return flag;
}

void Follow(void)
// построить множество follow_1
{
int flag,i,j,k;
// очищаем массив для формирования пар символов:
for ( i=0; i<MaxSym; i++)
      for ( j=0; j<MaxTerm; j++) fw[i][j]=0;

fw[left[0]-(int)'A'][(int)'#']=1;
             // аксиома - первый символ при вводе правил
        // # - символ-ограничитель
SetNew(left[0],'#');       // начальная пара S# построена

flag=1;
while (flag)
   {
```

```
   flag=0;
   for ( i=0; i<MaxSym; i++)            // i - индекс первого нетерминала
      for ( j=0; j<MaxTerm; j++)        // j - индекс второго символа
      if (fw[i][j]!='\0')
      k = SetNew(i+'A',j);
      flag=flag || k;
      }
}

int main(int argc, char * argv[])
{
if (argc<=1) GetData("input.txt");    // без параметра - файл по умолчанию
else GetData(argv[1]);        // ввести данные
Follow();
PrintPar();           // для демонстрационных целей - вывод пар символов
PrintFollow();
return 0;
}
```

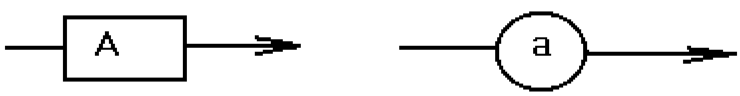
5.5 Программирование синтаксических диаграмм

Каждой синтаксической диаграмме соответствует отдельная процедура или функция программы синтаксического анализатора. Эта функция не содержит параметров, если требуется только синтаксический контроль исходной цепочки. Она может содержать параметры, если помимо синтаксического контроля выполняются другие действия: семантический контроль, интерпретация, компиляция. В главной программе транслятора вызов блока синтаксического анализатора реализуется вызовом той функции, которая соответствует аксиоме.

Рассмотрим программирование элементов синтаксической диаграммы в зависимости от ее структуры.

5.5.1 Простой нетерминальный или терминальный блок

Рассмотрим сначала самый простой элемент синтаксической диаграммы — один нетерминальный блок.



По определению синтаксической диаграммы каждому нетерминалу ставится в соответствие одна синтаксическая диаграмма, а каждой диаграмме соответствует одна функция синтаксического анализатора. Тогда реализация нетерминального элемента диаграммы заключается в вызове соответствующей функции:

```
A(); // вызов функции, соответствующей диаграмме A
```

Если очередной конструктивный элемент синтаксической диаграммы — терминальный символ, то необходимо отсканировать очередную лексему исходного модуля и проверить, действительно ли отсканированный символ является тем символом, который должен находиться в исходном модуле в соответствии с синтаксисом языка. Проверка на совпадение для всех без исключения лексических единиц может производиться по типу выделенной лексемы.

```
int t;   // локальная переменная - тип лексемы
LEX lex; // локальная переменная - изображение лексемы
   ...
t = Scaner (lex);
if ( t == <тип a> )
{<возможная семантическая обработка a> }
    else {PrintError(...);        //ошибка - ожидался символ a
         }
```

5.5.2 Ветвление

Как мы уже выяснили ранее, разметка параллельных ветвей заключается в построении функции $first_k(...)$ для этих ветвей. Следовательно, прежде чем перейти к анализу очередного фрагмента исходного модуля, надо определить, какй именно ветви соответствует этот фрагмент. Это значит, что требуется предварительное сканирование k очередных лексических единиц и сопоставление их с можеством $first_k(...)$. Тем самым мы однозначно определим ту ветвь, по которой требуется выполнять дальнейшую обработку текста.

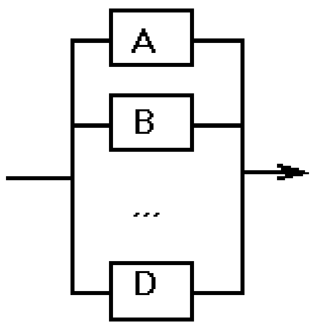


Рис. 5.25: Ветвление

Если каждая параллельная ветвь A, B, ..., D начинается терминалом (см. рис. 5.25.) и $k = 1$, то соответствующий фрагмент синтаксического анализатора имеет вид:

```
t=Scaner(lex); // отсканировали очередной терминал
if ( t== <first A> )    { < реализация ветви A >}
else if (t==<first B>)  { < реализация ветви B >}
else
   ...
else if (t==<first D>)  { < реализация ветви D >}
```

```
else {PrintError(...);
     // ошибка - неверный символ или неверная
     // конструкция, имя которой соответствует СД
}
```

Однако, если каждая из этих ветвей начинается нетерминалом, то внутренняя реализация соответствующей диаграммы выполняется в соответствии с ее структурой и независимо от всех предшествующих действий. Это означает, что предварительно отсканированные символы нужно "как бы вернуть на место". Для этого достаточно просто запомнить положение указателя перед началом такого предварительного сканирования, а затем вернуть его на исходную позицию. Будем использовать для этой цели специальные функции $GetUK()$ и $PutUK(int)$, которые внесем в модуль $Scaner, cpp$:

```
int GetUK(void)
{ return uk; }

void PutUK(int i)
{ uk = i; }
```

Если Ваша программа выдачи сообщения об ошибке $PrintError()$ выдает позицию в тексте ошибочного символа, то при реализации функций $GetUK()$ и $PutUK(int)$ следует также предусмотреть сохранение и восстановление не только UK, но и соответствующих переменных строки и позиции в строке.

Итак, если хотя бы одна ветвь начинается нетерминалом, необходимо предусмотреть последующую обработку этого нетерминала с соответствующего начального символа. Для этого перед вызовом сканера необходимо запомнить указатель uk в локальной рабочей области, а затем после сканирования восстановить его либо сразу за вызовом сканера, либо восстановление предусмотреть в каждой ветви, начинающейся нетерминалом:

```
int uk1; // локальная переменная
uk1 = GetUK();   t= Scaner (lex);   PutUK(uk1);
```

Дальнейший фрагмент программы реализации ветвления совпадает с предшествующим вариантом.

5.5.3 Циклы

Рассмотрим сначала цикл, в котором некоторая конструкция повторяется хотя бы один раз (см. рис. 5.26(a)).

После обработки конструкции A необходимо убедиться в необходимости продолжения цикла, для чего необходим анализ последующего текста. Это требует сканирования очередного символа. Тогда после завершения цикла следует восстановить указатель uk в предшествующей позиции. Такое восстановление делается в цикле при условии, что ветвь A начинается нетерминалом:

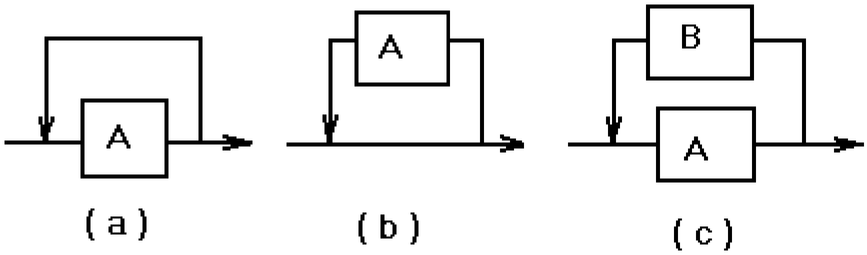


Рис. 5.26: Циклические синтаксические диаграммы

```
do
   {
   < реализация ветви A >
   uk1 = GetUK();  t= Scaner (lex);  PutUK(uk1);
   } while ( t == <first(A)> );
```

При терминальном начале ветви A программа несколько изменяется:

```
t= Scaner (lex);
do
   {
   < реализация ветви A без сканирования первого символа>
   uk1 = GetUK();  t= Scaner (lex);
   } while ( t == <first(A)> );
PutUK(uk1);
```

Рассмотрим теперь конструкцию, в которой на выполнение цикла наложено некоторое предусловие (см. рис. 5.26(b)). Если ветвь A начинается нетерминалом, то фрагмент программы имеет вид:

```
uk1 = GetUK();  t= Scaner (lex);  PutUK(uk1);
while ( t == <first(A)> )
      {
      < реализация ветви A >
      uk1 = GetUK();  t= Scaner (lex);  PutUK(uk1);
      }
```

Восстановление выносится из цикла при терминальном начале ветви A:

```
uk1 = GetUK();  t= Scaner (lex);
while ( t == <first(A)> )
      {
      < реализация ветви A без сканирования первого символа>
```

```
      uk1 = uk;  t= Scaner (lex);
      }
PutUK(uk1);
```

Иногда встречаются синтаксические конструкции, в которой выполнение цикла реализовано методом повторения фрагмента A с разделителем B. Рассмотрим синтаксичскую диаграмму для распознавания последовательности $ABAB...ABA = A(BA)^*$ (см. рис. 5.26(c)). Если B представляет собой единственный терминал, то конструкция программируется аналогично варианту (a) циклической диаграммы:

```
do
   {
   < реализация ветви A>
   uk1 = GetUK();  t= Scaner (lex);
   } while ( t == <first(B)> );
PutUK(uk1);
```

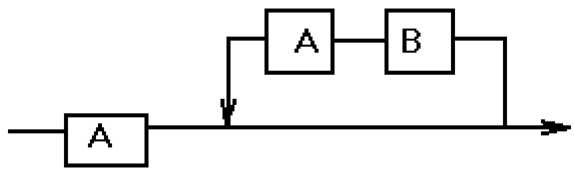


Рис. 5.27: Преобразование циклической синтаксические диаграммы рис. 5.26(c)

Анализ структуры программы показывает, что если B представляет собой единственный терминальный символ, то имеет смысл использовать такую конструкцию синтаксической диаграммы только при условии, если B не несет семантической нагрузки и, следовательно, в текст программы не придется встраивать специальные вызовы семантических подпрограмм. Если B — нетерминал, или состоящая из нескольких элементов конструкция, или элемент B несет семантическую нагрузку, то в целях получения более компактной программы желательно преобразовать диаграмму к виду, представленному на рис. 5.27, реализация которого существенно проще:

```
< реализация ветви A >
uk1 = GetUK();  t= Scaner (lex);  PutUK(uk1);
while ( t == <first(B)> )
      {
      < реализация ветви B >
      < реализация ветви A >
      uk1 = GetUK();  t= Scaner (lex);  PutUK(uk1);
      }
```

В качестве примера рассмотрим программу для диаграммы A из примера раздела 3.2 (см. рис. 5.16):

```
void A(void)
{
int t,uk1; LEX l;
E();
uk1=GetUK(); t=Scaner(l);
while ( (t==MULT) || (t==DIV) )
      {
      E();
      uk1=GetUK(); t=Scaner(l);
      }
PutUK(uk1);
}
```

5.5.4 Сообщения об ошибках

Рассмотренные нами алгоритмы синтаксического анализа реализуют правильный разбор исходной цепочки до первой возникшей ошибки. Сразу следует предостеречь от попытки продолжить анализ после обнаружения ошибки. Если в программе анализатора после обнаружения ошибки не будут выполнены никакие специальные действия по обработке ошибки и ее нейтрализации, то такой транслятор будет выдавать большое число так называемых *наведенных ошибок.*

Те сообщения об ошибках, которые мы сформулировали при каждом неверном символе, фактически означают, что мы предполагаем возникновение ошибки именно из-за символа, нарушающего синтаксис. Предложенные нами формулировки означают

Ожидался символ A. Вместо этого обнаружен символ B.

Сообщение об ошибке могут быть выражено и другими словами, но в приведенной форме по существу суммирована вся информация, понятная пользователю и точно локализующая место ошибки. Сообщение об ошибке должно сопровождаться указанием на ошибочный символ. В простейшем случае это номер строки и либо номер позиции в строке, либо изображение самого ошибочного символа.

Мы не проводим глубокого анализа текста в каждой конкретной синтаксической диаграмме и основываем сообщение об ошибке только на основе первого очередного терминального символа.

Опыт показывает, что часто в различных фрагментах программы синтаксического анализатора сообщения об ошибочных символах могут совпадать. Поэтому можно предложить два варианта реализации программы $PrintError(...)$.

Первый из них основан на том непреложном факте, что программа синтаксического анализатора должна быть простой и понятной. Известно, что хорошая программа должна быть снабжена комментариями. Если параметром функции $PrintError(...)$ является строка сообщения об ошибке, то оператор вызова этой функции одновременно является комментарием в программе синтаксического анализатора. Текст программы функции $PrintError(...)$ содержит всего несколько операторов:

```
void PrintError(char * err, LEX l)
{
```

```
printf("Строка %d позиция %d: %s %s\n",str,pos,err,l);
                // выдаем местоположение ошибки, текст сообщения
                // об ошибке err и ошибочный символ l
FinalWork();    // действия при завершении программы
exit(1);
}
```

Здесь функция $FinalWork()$ предназначена для специальных действия, которые нужно выполнить при завершении программы. Это может быть освобождение выделенной памяти, закрытие открытых файлов и т.п. Например, при трансляции оператора присваивания может быть выдана ошибка

```
PrintError("Ожидался символ присваивания ", "");,
```

а при сканировании неверного символа lex — ошибка

```
PrintError("Неверный символ ", lex);,
```

Другой вариант реализации функции $PrintError(...)$ основан на использовании номеров ошибок. Для этого можно использовать тип ошибок, например,

```
enum EROR {IDENT_ABSENT=1,    CONST_ABSENT,       SIGN_ABSENT,
   ...
           };
char Mess[MaxErr][MaxErrLen]={"Ожидался идентификатор",
                              "Ожидалась константа",
                              "Отсутствует знак операции",...};
```

Тогда в функции $PrintError(int\ n)$ нужно просто выдать текст сообщения, который находится в $Mess[n-1]$.

5.6 Контрольные вопросы к разделу 3

1. Дайте определение синтаксической диаграммы.

2. Как обозначаются терминальные и нетерминальные символы в синтаксической диаграмме?

3. Пусть для нетерминального символа A имеется несколько правил КС–грамматики

$$A \rightarrow \phi_1|\phi_2|...|\phi_k, \phi_i \in (V_T \cup V_N)^*.$$

Постройте соответствующую синтаксическую диаграмму.

4. Когда используется подстановка диаграммы в диаграмму? Как она выполняется?

5. Как устраняется левая рекурсия в синтаксических диаграммах?

6. Как устраняется правая рекурсия в синтаксических диаграммах?

7. Как выполняется вынесение общих множителей в синтаксических диаграммах?

8. Зависит ли вынесение общих множителей в синтаксической диаграмме от рекурсивной структуры этой диаграммы? Как проявляется эта зависимость?

9. Дайте формальное определение функции $first_k(Y)$. Что означает эта функция на неформальном уровне?

10. Дайте формальное определение функции $last_k(Y)$. Что означает эта функция на неформальном уровне?

11. Дайте формальное определение функции $follow_k(Y)$. Что означает эта функция на неформальном уровне?

12. Как размечаются точки ветвления в синтаксических диаграммах?

13. Как определяется условие выполнения цикла в синтаксических диаграммах?

14. Опишите алгоритм построения функции $first_k(Y)$.

15. Опишите алгоритм построения функции $last_k(Y)$.

16. Опишите алгоритм построения функции $follow_k(Y)$.

17. Какие методы разрешения неоднозначностей при реализации ветвлений в синтаксических диаграммах Вы знаете?

18. Как выполняется программирование синтаксических диаграмм?

19. Какие условия нужно проверить после разметки ветвей синтаксической диаграммы? Всегда ли размеченная диаграмма пригодна для программирования?

20. Что означает число k в функциях $first_k(Y)$, $follow_k(Y)$, $last_k(Y)$? Как это число влияет на скорость работы компилятора?

5.7 Тесты для самоконтроля к разделу 3

1. Какие преобразования синтаксических диаграмм применяются с целью построения эффективной программы синтаксического анализатора? Укажите такие преобразования из предлагаемого списка:

1) устранение левой рекурсии;
2) устранение правой рекурсии;
3) устранение центральной рекурсии;
4) вынесение общих множителей;
5) подстановка диаграммы в диаграмму.

Варианты ответов:

а) Все перечисленные преобразования.
б) Все, кроме 1.
в) Все, кроме 2.
г) Все, кроме 3.
д) Все, кроме 4.
е) Все, кроме 5.
б) Все, кроме 1.

Правильный ответ: г.

2. Даны правила грамматики

$$A \to Ab|c,$$

Укажите пример программы, которая правильно анализирует конструкцию A.

Варианты ответов:

```
a)  // ************ Диаграмма для A -> Ab | c
void A(void)
{
TypeLex l; int t,uk1;
t=Scaner(l);
if (t==first(A))
```

```
   {
   A();
   t=Scaner(l);
   if (t!=TYPE_b) PrintError("Неверный символ ", l);
   }
else
   if (t!=TYPE_c) PrintError("Неверный символ ", l);
}
```

```
б)  // ************ Диаграмма для A -> Ab | c
TypeLex l; int t,uk1;
void A(void)
{
t=Scaner(l);
if (t==first(A))
   {
   A();
   t=Scaner(l);
   if (t!=TYPE_b) PrintError("Неверный символ ", l);
   }
else
   if (t!=TYPE_c) PrintError("Неверный символ ", l);
}
```

```
в)  // ************ Диаграмма для A -> Ab | c
TypeLex l; int t,uk1;
void A(void)
{
t=Scaner(l);
if (t!=TYPE_c)  PrintError("Неверный символ ", l);
do
   {
   t=Scaner(l);
   } while (t==TYPE_b);
}
```

```
г)  // ************ Диаграмма для A -> Ab | c
void A(void)
{
TypeLex l; int t,uk1;
t=Scaner(l);
if ( (t!=TYPE_c) or (t!=TYPE_b) ) PrintError("Неверный символ ", l);
do
   {
   t=Scaner(l);
   } while (t==TYPE_b);
}
```

```
д)  // ************ Диаграмма для A -> Ab | c
void A(void)
{
```

```
TypeLex l; int t,uk1;
t=Scaner(l);
if (t!=TYPE_c)    PrintError("Неверный символ ", l);
while (t==TYPE_b)
   {
   t=Scaner(l);
   }
}
```

е) // ************ Диаграмма для A -> Ab | c

```
void A(void)
{
TypeLex l; int t,uk1;
t=Scaner(l);
if ((t!=TYPE_c) or (t!=TYPE_b)) PrintError("Неверный символ ", l);
while (t==TYPE_b)
   {
   uk1=GetUK();  t=Scaner(l);
   }
PutUK(uk1);
}
```

ж) // ************ Диаграмма для A -> Ab | c

```
TypeLex l; int t,uk1;
void A(void)
{
t=Scaner(l);
if (t!=TYPE_c)
   if (t!=TYPE_b) PrintError("Неверный символ ", l);
while (t==TYPE_b)
   {
   uk1=GetUK();  t=Scaner(l);
   }
PutUK(uk1);
}
```

з) // ************ Диаграмма для A -> Ab | c

```
TypeLex l; int t,uk1;
void A(void)
{
t=Scaner(l);
if ((t!=TYPE_c) or (t!=TYPE_b)) PrintError("Неверный символ ", l);
uk1=GetUK();  t=Scaner(l);
while (t==TYPE_b)
   {
   uk1=GetUK();  t=Scaner(l);
   }
PutUK(uk1);
}
```

и) // ************ Диаграмма для A -> Ab | c

```
TypeLex l; int t,uk1;
void A(void)
{
t=Scaner(l);
if (t!=TYPE_c) PrintError("Неверный символ ", l);
uk1=GetUK();  t=Scaner(l);
while (t==TYPE_b)
   {
   uk1=GetUK();  t=Scaner(l);
   }
PutUK(uk1);
}
```

```
к)  // ************ Диаграмма для A -> Ab | c
TypeLex l; int t,uk1;
void A(void)
{
t=Scaner(l);
if (t!=TYPE_c) PrintError("Неверный символ ", l);
do
   {
   uk1=GetUK();  t=Scaner(l);
   } while (t==TYPE_b);
PutUK(uk1);
}
```

Правильный ответ: и.

3. Дана грамматика

$$\begin{aligned} G: \quad & S \to AB \\ & A \to aADb|\varepsilon \\ & D \to Aab|c \\ & B \to cBa|\varepsilon. \end{aligned}$$

Вычислить функцию $first_1(S)$.
Варианты ответов:
а) $\{a\}$;
б) $\{c\}$;
в) $\{a, c\}$;
г) $\{a, c, \varepsilon\}$;
д) $\{\varepsilon\}$;
е) $\{a, c, \natural\}$;
ж) $\{a, c, \natural, \varepsilon\}$;.
Правильный ответ: г.

4. 3. Дана грамматика

$$\begin{aligned} G: \quad & S \to AB \\ & A \to aADb|d \\ & D \to Aab|\varepsilon \\ & B \to cBa|\varepsilon. \end{aligned}$$

Вычислить функцию $follow_1(A)$.
Варианты ответов:

а) $\{a, c\}$;
б) $\{b, c\}$;
в) $\{a, c, \natural\}$;
г) $\{a, c, \varepsilon\}$;
д) $\{a, b, c, d, \varepsilon\}$;
е) $\{a, b, c, d, \natural\}$;
ж) $\{a, b, c, \varepsilon\}$;.
з) $\{a, b, c, \natural, \varepsilon\}$;.

Правильный ответ: е.

5. Даны правила грамматики

$$A \to aADb|d$$

Какая конструкция соответствует грамматическому разбору нетерминала A?

Варианты ответов:
а) цикл с предусловием;
б) цикл с постусловием;
в) оператор ветвления;
г) линейная конструкция.

Правильный ответ: в.

5.8 Упражнения к разделу

5.8.1 Задание

Цель данного задания – реализация программы синтаксического анализатора методом рекурсивного спуска. В качестве грамматики Вам предлагается взять грамматику из упражнения к главе 1. Эта грамматика описывает весь язык полностью без выделения лексического и синтаксического уровней. Лексический уровень грамматики Вы уже реализовали, выполняя упражнение к главе 2. Теперь Вы должны выделить синтаксический уровень грамматики и реализовать программу синтаксического анализа. Выполнение задания предусматривает следующий порядок работ.

1. Выписать синтаксический уровень КС–грамматики Вашего задания. Для наглядности обозначить все нетерминалы большими латинскими буквами. Для каждого такого нетерминала оставить комментарий — его назначение в грамматике и определяемую конструкцию.

2. Построить синтаксические диаграммы КС–грамматики Вашего задания.

3. Преобразовать построенные синтаксические диаграммы:

- устранить левую и правую рекурсию;

- вынести левые и правые множители;

- выполнить подстановку диаграммы в диаграмму, если в результате такой подстановки уменьшится общая сложность конструкции.

4. При устранении рекурсии учитывать семантическую нагрузку, которую несут отдельные лексемы и конструкции. Не объединять ветви цикла с находящиейся перед циклом конструкцией, если конструктивные элементы цикла разделяются знаком, имеющим семантическую нагрузку.

5. Построить функции $first$ для всех нетерминалов Вашей грамматики.

6. Построить функции $follow$.

7. Разметить ветви синтаксических диаграмм с использованием построенных функций $first$ и $follow$.

8. Проверить однозначность переходов в диаграммах по точкам ветвления. Если переходы неоднозначны, то выбрать один из вариантов разрешения каждого конфликта:

- преобразовать грамматику и синтаксические диаграммы;

- увеличить длину анализируемого контекста в точках ветвления;

- принять некоторые соглашения, однозначно определяющие действия в точке ветвления (например, по аналогии с условием, при котором $else$ всегда соответствует последнему if).

9. Каждой полученной Вами синтаксической диаграмме поставить в соответствие процедуру (функцию) без параметров. В силу рекурсивного характера совокупности синтаксических диаграмм предусмотреть предварительное объявление каждой функции, например, в модуле *Diagram.hpp*.

10. Написать тело каждой функции, соответствующей синтаксической диаграмме.

11. В главной программе предусмотреть вызов функции, соответствующей аксиоме.

12. Дополнить проект, построенный Вами при выполнении задания к главе 2, программным модулем, в котором содержится реализация функций всех синтаксических диаграмм, например, *Diagram.cpp*.

13. Дополнить набор сообщений об ошибках, которые выдаются программой $PrintError()$.

14. Отладить программу на правильных и неправильных синтаксических конструкциях в исходном модуле.

5.8.2 Пример выполнения задания

Выпишем синтаксический уровень КС–грамматики из примера выполнения задания к главе 1. На лексическом уровне грамматики находятся все символы, которые выделяются сканером. Таблица лексических единиц была нами построена в примере к главе 2. Сложные лексемы, правила построения которых следует удалить из синтаксического уровня КС–грамматики, — это в соответствии с построенной таблицей лексем следующие понятия :

а) идентификаторы,

б) строковые константы,

в) константы целые,

г) константы вещественные с точкой.

д) константы в экспоненциальной форме.

Все остальные лексические единицы используются в грамматике в форме изображения и, следовательно, не были определены специальными грамматическими правилами. В результате получим грамматику:

G_J :

< программа >→	< **SCRIPT language =″ JavaScript″** >
	<! − −
	< описания >
	−− >
	< **/SCRIPT** >
< описания >→	< описания > < одно описание > |ε
< одно описание >→	< данные > | < функция >
< данные >→	**var** < список > ;
< список >→	< список > , < переменная > | < переменная >
< переменная >→	< идентификатор > |
	< идентификатор > = < выражение >
< функция >→	**function**(< список >) < составной оператор >
< составной оператор >→	{< операторы и описания >}
< операторы и описания >→	< операторы и описания >< данные > |
	< операторы и описания >< оператор > |ε
< оператор >→	< присваивание > ; | < составной оператор >
	< вызов функции > | < for > | < if > | ;
< for >→	**for**(< присваивание >; < выражение >;
	< присваивание >) < оператор >
< if >→	**if**(< выражение >) < оператор > |
	if(< выражение >) < оператор > **else** < оператор >
< присваивание >→	< имя > = < выражение >
< имя >→	< имя > . < идентификатор > | < идентификатор >

< выражение >→	< выражение > > < слагаемое > |
	< выражение > >= < слагаемое > |
	< выражение > < < слагаемое > |
	< выражение > <= < слагаемое > |
	< выражение > == < слагаемое > |
	< выражение > ! = < слагаемое > |
	+ < слагаемое > |
	− < слагаемое > |
	< слагаемое >
< слагаемое >→	< слагаемое > + < множитель > |
	< слагаемое > − < множитель > |
	< множитель >
< множитель >→	< множитель > ∗ < эл.выр. > |
	< множитель > / < эл.выр. > |
	< эл.выр. >
< эл.выр. >→	< имя > | < константа > |
	< вызов функции > |(< выражение >)
< вызов функции >→	< идентификатор > (< параметры >)|
	< идентификатор > ()
< параметры >→	< параметры > , < выражение > | < выражение >
< константа >→	< конст.целая > | < конст.веществ. > |
	< конст.экспон. > | < конст.символьн. >

Для наглядности и простоты обозначим маленькими латинскими буквами a, c_1, c_2, c_3, c_4 терминальные символы синтаксического уровня — соответственно идентификаторы и перечисленные выше константы четырех типов. Обозначим большими латинскими буквами нетерминалы и занесем эти обозначения в таблицу:

Понятие	Новое обозначение
$<$ программа $>$	S
$<$ описания $>$	T
$<$ одно описание $>$	W
$<$ данные $>$	D
$<$ список $>$	Z
$<$ переменная $>$	I
$<$ выражение $>$	V
$<$ функция $>$	F
$<$ составной оператор $>$	Q
$<$ оператор $>$	O
$<$ операторы и описания $>$	K
$<$ присваивание $>$	P
$<$ вызов функции $>$	H
$< for >$	U
$< if >$	M
$<$ имя $>$	N
$<$ слагаемое $>$	A
$<$ множитель $>$	B
$<$ эл.выр. $>$	E
$<$ параметры $>$	X
$<$ константа $>$	C

В результате получим грамматику

$$
\begin{array}{lll}
G_J: & S \rightarrow & <\textbf{SCRIPT language} =''\ \textbf{JavaScript}''> \\
 & & <!--T--> \\
 & & </\textbf{SCRIPT}> \\
 & T \rightarrow & T\ W|\varepsilon \\
 & W \rightarrow & D|F \\
 & D \rightarrow & \textbf{var}\ Z; \\
 & Z \rightarrow & Z,\ I|I \\
 & I \rightarrow & a\ |\ a{=}V \\
 & F \rightarrow & \textbf{function}\ (Z)\ Q \\
 & Q \rightarrow & \{K\} \\
 & K \rightarrow & KD|KO|\varepsilon \\
 & O \rightarrow & P\textbf{;}\ |Q|H|U|M|\ ; \\
 & U \rightarrow & \textbf{for}\ (P;V;P)\ O \\
 & M \rightarrow & \textbf{if}\ (V)\ O|\textbf{if}\ (V)\ O\ \textbf{else}\ O \\
 & P \rightarrow & N{=}V \\
 & N \rightarrow & N.a|a \\
 & V \rightarrow & V{>}A|V{>=}A|V{<}A|V{<=}A|V{==}A|V!=A|{+}A|{-}A|A \\
 & A \rightarrow & A{+}B|A{-}B|B \\
 & B \rightarrow & B{*}E|B/E|E \\
 & E \rightarrow & N|C|H|(V) \\
 & H \rightarrow & a(X)|a() \\
 & X \rightarrow & X,\ V|V \\
 & C \rightarrow & c_1|c_2|c_3|c_4
\end{array}
$$

Синтаксические диаграммы строятся достаточно просто, поэтому отдельно процесс построения мы рассматривать не будем. Отметим лишь, что диаграммы для нетерминалов S, D, F, Q, U, P представляют собой простые последовательные структуры. Диаграммы для V, A, B, Z, T, X, K требуют устранения рекурсии. Для простых диаграмм C, N, X была выполнена подстановка в диаграммы более высокого уровня.

Перейдем к программированию синтаксического уровня грамматики. Будем дополнять проект, построенный при выполнении задания к главе 2. Введем в проект модуль *Diagram.cpp*, а в *Diagram.hpp* укажем заголовки функций, которые реализуют соответствующие синтаксические диаграммы. В качестве рекомендации, для упрощения процесса написания программы и ее отладки, можно посоветовать поставить перед телом каждой функции комментарий, в котором представлена структура диаграммы. Для опытного программиста это могут быть непосредственно правила КС–грамматики. При начальном опыте программирования синтаксических анализаторов желательно иметь наглядное представление вида диаграммы.

```
//********************************************
// Diagram.hpp - класс  синтаксических диаграмм
//********************************************
#ifndef __DIAGRAM
#define __DIAGRAM
class TDiagram
{
private:
   TScaner *sc;

public:
   TDiagram(TScaner * s) {sc=s;}
   ~TDiagram(){}
   void S(); // программа
   void Q(); // операторы и описания
   void T(); // описания
   void W(); // одно описание
   void D(); // данные
   void Z(); // список
   void V(); // выражение
   void F(); // функция
   void K(); // составной оператор
   void O(); // оператор
   void P(); // присваивание
   void H(); // вызов функции
   void U(); // оператор for
   void M(); // оператор if
   void N(); // имя
   void A(); // слагаемое
   void B(); // множитель
   void E(); // элементарное выражение
};
#endif
```

```
//*******************************************************************
// Diagram.cpp -  функции, реализующие синтаксические диаграммы
//*******************************************************************
#include "defs.hpp"
#include "Scaner.hpp"
#include "diagram.hpp"

void TDiagram::S()
// программа
//  < SCRIPT language = "javaScript" > <!--  T  --> </ SCRIPT >
{
TypeLex l; int t;
t=sc->Scaner(l);
if (t!=TLT) sc->PrintError("ожидался символ <",l);
t=sc->Scaner(l);
if (t!=TScript) sc->PrintError("ожидался символ script",l);
t=sc->Scaner(l);
if (t!=TLang) sc->PrintError("ожидался символ language",l);
t=sc->Scaner(l);
if (t!=TSave) sc->PrintError("ожидался знак =",l);
t=sc->Scaner(l);
if (t!=TConsChar) sc->PrintError("ожидалась символьная константа",l);
t=sc->Scaner(l);
if (t!=TGT) sc->PrintError("ожидался символ >",l);
t=sc->Scaner(l);
if (t!=TCom1) sc->PrintError("ожидался символ <!--",l);
T();
t=sc->Scaner(l);
if (t!=TCom2) sc->PrintError("ожидался символ -->",l);
t=sc->Scaner(l);
if (t!=TTegEnd) sc->PrintError("ожидался символ </",l);
t=sc->Scaner(l);
if (t!=TScript) sc->PrintError("ожидался символ script",l);
t=sc->Scaner(l);
if (t!=TGT) sc->PrintError("ожидался символ >",l);
}


void TDiagram:: Q()
// Составной оператор   -----
// ----------- { -------| K |---------- } --------->
//                      -----
{
TypeLex l; int t;
t=sc->Scaner(l);
if (t!=TFLS) sc->PrintError("ожидался символ {",l);
K();
t=sc->Scaner(l);
if (t!=TFPS) sc->PrintError("ожидался символ }",l);
```

```
}


void TDiagram:: T()
//                    -----
// описания      -----| W |---- var
//               |    -----   | function
//  -------------.--------------------->
{
TypeLex l; int t,uk1;
uk1=sc->GetUK(); t=sc->Scaner(l); sc->PutUK(uk1);
while((t==TVar) || (t==TFunct) )
  {
  W();
  uk1=sc->GetUK(); t=sc->Scaner(l); sc->PutUK(uk1);
  }
}


void TDiagram:: W()
// одно описание
//            var  -----
//           ------| D |----
//           |     -----   |
//     ------|     -----   |-------->
//           ------| F |----
//     function    -----
{
TypeLex l; int t,uk1;
uk1=sc->GetUK(); t=sc->Scaner(l); sc->PutUK(uk1);
if (t==TVar) D();
else F();
}


void TDiagram:: D()
// данные            -----
//   ------ var -----| Z |--- ; ---->
//                   -----
{
TypeLex l; int t;
t=sc->Scaner(l);
if (t!=TVar) sc->PrintError("ожидался символ var",l);
Z();
t=sc->Scaner(l);
if (t!=TTZpt) sc->PrintError("ожидался символ ;",l);
}


void TDiagram:: Z()
// список
```

```
//          ------------ , -----------------
//          |                 -----        |
//          |        --- = ---| V |------  |
//  -------.-- a ---|        -----     |----------->
//                   --------------------
//
{
TypeLex l; int t, uk1;
do {
   t=sc->Scaner(l);
   if (t!=TIdent) sc->PrintError("ожидался идентификатор",l);
   uk1=sc->GetUK();   t=sc->Scaner(l);
   if (t==TSave)
     {
     V();
     uk1=sc->GetUK(); t=sc->Scaner(l);
     }
   } while(t==TZpt);
sc->PutUK(uk1);
}
```

```
void TDiagram:: V()
// выражение                           ---- != ------
//                                     |--- == -----|
//                                     |--- <= -----|
//                              -----  |---- < -----|
//                     ---------| A |--|--- >= -----|
//    --  + --  -----  |        -----  ----- > -----|
// ---|------|--| A |--.------------------------------------->
//    --  - --  -----
{
TypeLex l; int t,uk1;
uk1=sc->GetUK(); t=sc->Scaner(l);
if ( (t!=TPlus) && (t!=TMinus) )sc->PutUK(uk1);
A();
uk1=sc->GetUK(); t=sc->Scaner(l);
while ((t<=TNEQ) && (t>=TLT)) // знаки сравнения стоят подряд
    {
    A();
    uk1=sc->GetUK(); t=sc->Scaner(l);
    }
sc->PutUK(uk1);
}
```

```
void TDiagram:: F()
```

```
// функция
//                                  -----
//                          --------| Z |------          -----
// ---- function --a--- ( --|       -----     |--- ) ----| Q |---->
//                          ------------------           -----
//
{
TypeLex l; int t, uk1;
t=sc->Scaner(l);
if (t!=TFunct) sc->PrintError("ожидался символ function",l);
t=sc->Scaner(l);
if (t!=TIdent) sc->PrintError("ожидался идентификатор",l);
t=sc->Scaner(l);
if (t!=TLS) sc->PrintError("ожидался символ (",l);
uk1=sc->GetUK(); t=sc->Scaner(l);
if (t!=TPS)  { sc->PutUK(uk1); Z(); t=sc->Scaner(l); }
if (t!=TPS) sc->PrintError("ожидался символ )",l);
Q();
}


void TDiagram:: K()
// Операторы и описания
//                    -----   var
//          ----------| D |-----------
//          |         -----          |
//          |         -----          |
//          |---------| O |----------|
//          |         -----          |
//  --------.---------------------------------->
//                                          }
{
TypeLex l; int t,uk1;
uk1=sc->GetUK(); t=sc->Scaner(l);  sc->PutUK(uk1);
while ( t!=TFPS)
     {
     if (t==TVar)  D();
else       O();
     uk1=sc->GetUK(); t=sc->Scaner(l);  sc->PutUK(uk1);
     }
}


void TDiagram:: O()
// оператор
//              a=    -----
//          ----------| P |-- ; ------
//          |         -----          |
//          |---------- ; -----------|
//          |  {      -----          |
```

```
//          |---------| Q |----------|
//          |         -----          |
//          |  a(     -----          |
//          |---------| H |----------|
//          |         -----          |
//          |  for    -----          |
//  --------|---------| U |----------|--------->
//          |         -----          |
//          |    if   -----          |
//          ----------| M |-----------
//                    -----
{
TypeLex l; int t,uk1;
uk1=sc->GetUK(); t=sc->Scaner(l);
if ( t==TTZpt)  return; // пустой оператор
if (t==TIf)  { sc->PutUK(uk1); M();  return;}
if (t==TFor) { sc->PutUK(uk1); U();  return;}
if (t==TFLS) { sc->PutUK(uk1); Q();  return;}
// остались H и P , анализируемые по first2
t=sc->Scaner(l);  sc->PutUK(uk1);
if (t==TSave)
     {
     P();
     t=sc->Scaner(l);
     if (t!=TTZpt) sc->PrintError("ожидался символ ;",l);
     }
     else     H();
}

void TDiagram:: P()
// присваивание
//
//      ---- .-----
//      |         |            -----
// -----.---  a ------ = ------| V |-------->
//                             -----
{
TypeLex l; int t, uk1;
do {
   t=sc->Scaner(l);
   if (t!=TIdent) sc->PrintError("ожидался идентификатор",l);
   uk1=sc->GetUK(); t=sc->Scaner(l);
   } while (t==TToch);
if (t!=TSave) sc->PrintError("ожидался знак =",l);
V();
}

void TDiagram:: H()
// вызов функции           ------------
//                         |   -----   |
```

```
//                      ---.---| V |-------
//                      |      -----      |
// -------- a ----- (---|                 |---- ) ----->
//                      -------------------
{
TypeLex l; int t, uk1;
t=sc->Scaner(l);
if (t!=TIdent) sc->PrintError("ожидался идентификатор",l);
t=sc->Scaner(l);
if (t!=TLS) sc->PrintError("ожидался символ (",l);
uk1=sc->GetUK(); t=sc->Scaner(l);
if (t==TPS) return; // нет параметров
do {
   V();
   uk1=sc->GetUK(); t=sc->Scaner(l);
   if (t!=TPS) sc->PutUK(uk1);
   } while (t!=TPS);
if (t!=TPS) sc->PrintError("ожидался знак )",l);
}


void TDiagram:: U()
// оператор for   ----     ----          ----       ----
// --- for -- ( --| P|-----| V|--- ;-----|P |-- ) --| O|----->
//               ----     ----          ----       ----
{
TypeLex l; int t;
t=sc->Scaner(l);
if (t!=TFor) sc->PrintError("ожидался символ for",l);
t=sc->Scaner(l);
if (t!=TLS) sc->PrintError("ожидался символ (",l);
P();
t=sc->Scaner(l);
if (t!=TTZpt) sc->PrintError("ожидался символ ;",l);
V();
t=sc->Scaner(l);
if (t!=TTZpt) sc->PrintError("ожидался символ ;",l);
P();
t=sc->Scaner(l);
if (t!=TPS) sc->PrintError("ожидался символ )",l);
O();
}


void TDiagram:: M()
// оператор if
//                                                    ----
//              ----        ----    ----- else -----|O |---
//  ---- if ( ---|V |--- ) --|O |----|                ----  |----->
//              ----        ----    ----------------------
```

```
{
TypeLex l; int t,uk1;
t=sc->Scaner(l);
if (t!=TIf) sc->PrintError("ожидался if )",l);
t=sc->Scaner(l);
if (t!=TLS) sc->PrintError("ожидался символ (",l);
V();
t=sc->Scaner(l);
if (t!=TPS) sc->PrintError("ожидался символ )",l);
uk1=sc->GetUK(); t=sc->Scaner(l);
if (t==TElse) O();
   else       sc->PutUK(uk1);
}

void TDiagram:: N()
// имя
//      ---- .-----
//      |         |
// -----.---  a ------->
{
TypeLex l; int t,uk1;
do {
   t=sc->Scaner(l);
   if (t!=TIdent) sc->PrintError("ожидался идентификатор",l);
   uk1=sc->GetUK(); t=sc->Scaner(l);
   } while (t==TToch);
sc->PutUK(uk1);
}

void TDiagram:: A()
// слагаемое
//                       -----  ----- + ------
//               ---------| B |--|           |
//        -----  |        -----  ----- - -----|
// --------| B |--.------------------------------------>
//        -----
{
TypeLex l; int t,uk1;
B();
uk1=sc->GetUK(); t=sc->Scaner(l);
while ((t==TPlus) || (t==TMinus))
    {
    B();
    uk1=sc->GetUK(); t=sc->Scaner(l);
    }
sc->PutUK(uk1);
}

void TDiagram:: B()
// множитель
```

```
//                          -----  ----- * ------
//                ---------| E |--|            |
//          -----  |        -----  ----- / -----|
// --------| E |--.-------------------------------------->
//          -----
{
TypeLex l; int t,uk1;
E();
uk1=sc->GetUK(); t=sc->Scaner(l);
while ((t==TDiv) || (t==TMult))
    {
    E();
    uk1=sc->GetUK(); t=sc->Scaner(l);
    }
sc->PutUK(uk1);
}

void TDiagram:: E()
// элементарное выражение
//           ----------- c1-----------
//           |---------- c2----------|
//           |---------- c3----------|
//           |---------- c4----------|
//           |   a.    -----         |
//           |---------| N |----------|
//           |         -----         |
//           |   a(    -----         |
//  --------|---------| H |----------|--------->
//           |         -----         |
//           |         -----         |
//           ----(-----| V |----- ) ---
//                     -----
{
TypeLex l; int t,uk1;
uk1=sc->GetUK(); t=sc->Scaner(l);
if ( (t==TConsChar) || (t==TConsInt)
   ||(t==TConsFloat) ||(t==TConsExp) ) return;
if (t==TLS)
    {
    V(); t=sc->Scaner(l);
    if (t!=TPS) sc->PrintError("ожидался символ )",l);
    return;
    }
// для определения N или  H нужно иметь first2
t=sc->Scaner(l);  sc->PutUK(uk1); // восстанавливается uk начальное
if (t==TLS) H();
     else   N();
}
```

В главной программе транслятора вызовем функцию $S()$, которая соответствует аксиоме грамматики. Может оказаться, что для исходного модуля, который поступил на вход программе анализатора, дерево разбора будет построено правильно, но при этом за концом транслируемой программы останется какой–то фрагмент. Этот фрагмент транслятор не станет обрабатывать. Для того, чтобы исключить такое явление, достаточно после выхода из функции $S()$ отсканировать очередную лексему. Если программа без ошибок, то этот символ – ограничитель конца $TEnd$.

```
//**************************************************************
//  Главная программа транслятора -  выполняется  синтаксический
//  анализ методом рекурсивного спуска  до первой ошибки
//**************************************************************
#include <stdio.h>
#include <string.h>
#include "defs.hpp"
#include "Scaner.hpp"
#include "diagram.hpp"

int main(int argc, char * argv[])
{
TScaner  * sc ;
      // ввести файл с исходным модулем:
if (argc<=1) sc = new TScaner("input.txt");//  файл по умолчанию
        else sc = new TScaner(argv[1]);    // задан файл

TDiagram *dg = new TDiagram(sc);
dg->S();

int type; TypeLex l;
type=sc->Scaner(l);
if (type==TEnd)   printf("Синтаксических ошибок не обнаружено. \n");
   else           sc->PrintError("Лишний текст в конце программы.","");
return 0;
}
```

Для отладки синтаксического анализатора необходимо сначала убедиться в том, что правильная структура исходного модуля обрабатывается Вашей программой верно. В отладочный текст транслируемой Вашим анализатором программы необходимо внести всевозможные конструкции каждого синтаксического фрагмента. Например, для проверки обработки оператора *var* необходимо использовать следующие конструкции:

- одна переменная без начальной инициализации;
- много переменных без начальной инициализации;
- одна переменная с начальной инициализацией;
- много переменных с начальной инициализацией;
- много переменных как с начальной инициализацией, так и без нее.

Пример отладочных данных, причем далеко не исчерпывающий все возможные ситуации, может иметь вид:

```
 <script  language="javascript">
```

```
<!--
var d=4; var ffff=34, ggg=5+6, hhh, ggggg;
var a,s,d,fd;
function rrr(){} // функция с пустым телом
var d;
var d=4; var ffff=34, ggg=5+6, hhh, ggggg;
   // описания с инициализацией  и без нее
function fun1(){}
function fun2(a,b,c,d){;;;;;;} // пустые операторы подряд
function fun3(a,b,c,d){a=3;}   // один оператор  в теле
function fun4(a,b,c,d){ a=3; var e,e,e,e;{}}
function fun5(a,b,c,d)  {
  var s,s,s;
  a=3; var e,e,e,e;
  {}
    // последовательность вложенных операторов for и  if:
  for ( i=0;    nu1<3-((((3-3-3-3))));   u2=9-(i)    )
   for (i=0;  j>=7;  i=1+i)
    if (a==7) for (j=0;j<=18; j=2+j)
       if (a==7) for (j=0;j<=18; j=2+j)
           if (a==7) for (j=0;j<=18; j=2+j)
                if (f!=0) j=5;
  {{{{ }}}}   // пустые вложенные операторы
}
 -->
</script >
```

Глава 6

КОНТЕКСТНЫЕ УСЛОВИЯ

Как уже отмечалось в разделе 3, программа является семантически правильной, если она соответствует контекстным условиям языка программирования. Контекстные условия контролируются семантическими подпрограммами, которые используют таблицы, содержащие семантическую информацию об объектах исходного модуля. Поэтому прежде, чем начинать программирование семантических подпрограмм, необходимо рассмотреть конструкцию таблиц, их содержимое и особенности реализации.

6.1 Структура таблиц компилятора

Структура и состав таблиц транслятора определяются языком программирования и зависят от типа проектируемого компилятора. Как правило, языки программирования содержат данные в виде констант, переменных, типов. Информация о таких данных различна, поэтому обычно выделяются следующие объекты транслируемой программы, контекстная информация о которых собирается в таблицах транслятора:

1) константы;
2) метки;
3) простые переменные;
4) массивы — данные и типы;
5) структуры — данные и типы;
6) процедуры и функции;
7) некоторые другие объекты в зависимости от языка программирования, например, классы и переменные различных пользовательских типов.

Общий подход к реализации таблицы определяется как необходимостью реализации универсального метода представления таблиц, так и стремлением получить максимально простые семантические подпрограммы. Если язык не допускает описание пользовательских типов данных, то можно формировать разные таблицы для объектов разных типов. Если язык программирования допускает пользовательские типы данных, то все объекты, в том числе и типы, заносятся в одну таблицу. Можно использовать две таблицы — объектов и типов, организация которых имеет вид, представленный на рис. 6.1.

Для того, чтобы стандартные типы данных языка программирования обрабатывались так же, как и любые пользовательские типы, в начале таблицы типов должно содержаться описание стандартных типов.

Если язык программирования допускает использование областей видимости, то

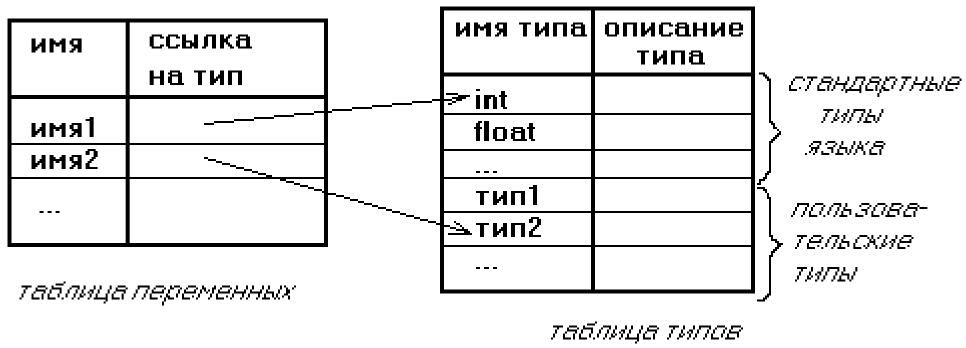


Рис. 6.1: Таблица переменных и таблица типов транслятора

структура таблиц должна поддерживать блочную структуру программы. В этом случае удобно формировать таблицу в виде бинарного дерева. По определению бинарного дерева каждая его вершина может иметь (или не иметь) левого и правого потомка, а каждая вершина, кроме корня, имеет родителя. Потомки каждого родителя упорядочены, так что левый потомок отличается от правого и потомки нельзя менять местами. Свойство упорядоченности позволяет сформировать зависимость описанных в программе данных так, что каждый элемент таблицы транслятора представляет собой вершину, левый потомок которой является соседом текущего уровня, а правый соответствует следующему уровню. Правые потомки могут быть у структур, а также у процедур и функций. При этом правые потомки процедур и функций представляют вложенные параметры и данные, причем первыми в цепочке расположены параметры.

На рисунке 6.2 приведен пример семантического дерева для программы, в которой имеются переменные, описнные на разном уровне вложенности. Для простоты реализации, как мы увидим в дальнейшем, кроме информационных узлов в семантическом дереве присутствуют пустые узлы, отмеченные на рисунке черным.

Древовидная таблица позволяет просто и эффективно отслеживать уровни вложенности описаний, отмечать параметры функций, осуществлять поиск данных в соответствии со структурой описания. В каждой точке транслируемой программы можно использовать только те объекты, которые находятся в древовидной таблице от узла текущего уровня вверх по дереву до корня включительно.

6.2 Информация в таблице компилятора

Итак, в соответствии с требованиями семантического контроля для реализуемого языка программирования разработчик выбрал способ представления таблиц компилятора. Как уже отмечалась ранее, это или единственная таблица, или несколько различных взаимосвязанных или несвязанных таблиц. Рассмотрим теперь информацию, которую необходимо хранить в таблицах для различных данных. В зависимости от языка программирования реализуются либо отдельные таблицы для каждых типов данных, либо гибридные таблицы, содержащие все данные, так, что каждый элемент таблицы — структура типа *union* всех допустимых описаний единичных типов данных.

В соответствии с тем, какие типы объектов могут использоваться в языке программирования, введем обозначение типа $TypeObject$, интегрирующего все семанти-

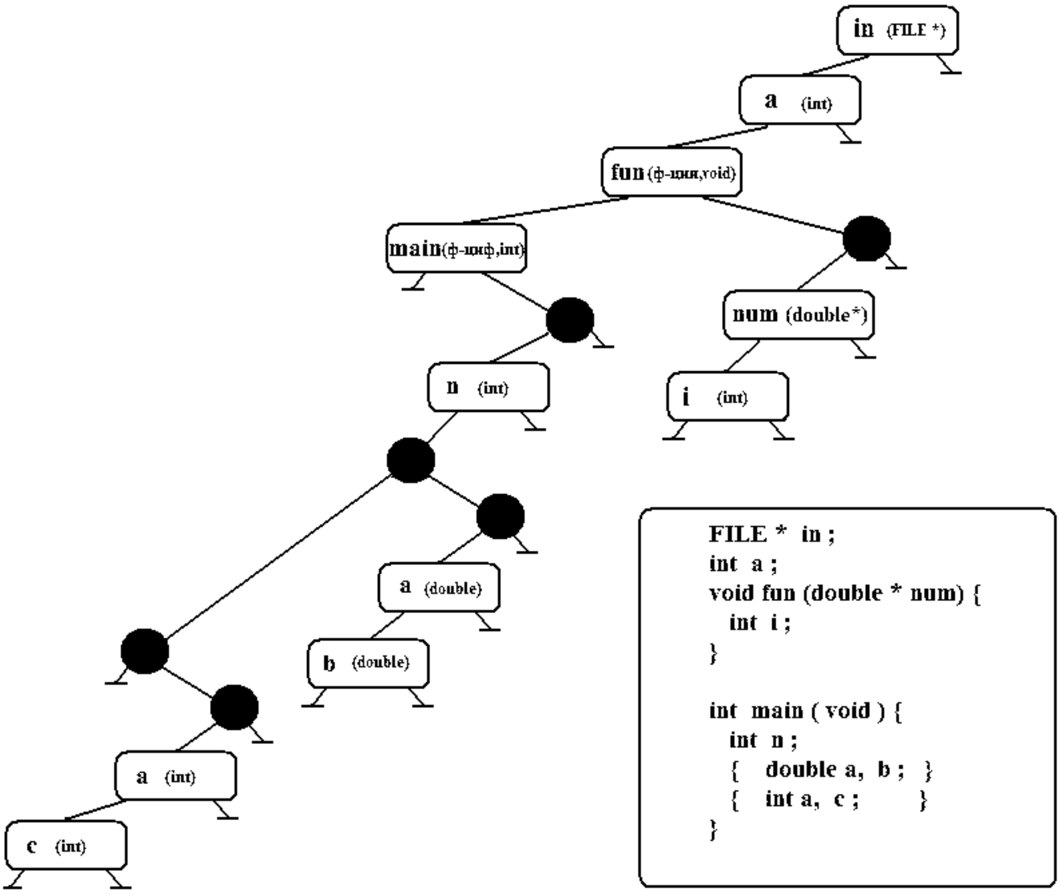


Рис. 6.2: Пример семантического дерева

ческие типы в транслируемой программе.

```
enum TypeObject {ObjConst=1,     // константа
                 ObjLabel,       // метка
                 ObjVar,         // простая переменная
                 ObjTypeVar,     // простой тип
                 ObjArray,       // массив
                 ObjTypeArray,   // тип массива
                 ObjStruct,      // структура
                 ObjTypeStruct,  // тип  структуры
                 ObjFunct};      // функция
```

В области данных для объекта любого типа будем хранить соответствующий признак этого объекта типа $TypeObject$.

6.2.1 Простые переменные

Для простых переменных обязательными полями в таблице являются идентификатор переменной и ее тип (или ссылка на тип). Кроме обязательных полей в таблице могут присутствовать некоторые дополнительные поля. Рассмотрим структуру дополнительной информации. Ее наличие определяется как особенностями языка программирования, так и типом проектируемого компилятора.

В некоторых языках программирования допускается описание переменной вместе с ее инициализацией. Например, в языке Си можно написать

```
int N=100, M=-345;
FILE *in =  fopen("input.txt","r");
FILE *out = fopen("output.txt","w");
float a[10]={1,2,3};
char t[100]="Начальная инициализация стороки\n";
```

Сведения о начальной инициализации можно хранить в специальном поле таблицы, хотя, как правило, эта информация является избыточной, потому что соответствующие операции присваивания встраиваются непосредственно в результирующую программу. Это хорошо заметно при пошаговом выполнении программы в среде программирования, когда в окне просмотра последовательно изменяются значения переменных в соответствии с порядком их инициализации.

Для простых переменных, как впрочем и для других объектов, для которых в области данных оттранслированной программы резервируется область памяти, имеется еще один параметр, который иногда хранится в таблице. Это относительный адрес в оттранслированной программе. Этот адрес необходим, если исходный модуль транслируется сразу в объектный код. Если выполняется интерпретация, то в указанной области расположено либо значение переменной, либо адрес, по которому оно хранится в процессе интерпретации. Если же трансляция осуществляется на некоторый промежуточный язык (например, на язык Ассемблера), то распределением памяти будет заниматься соответствующая программа перевода этого промежуточного кода в объектный код и, следовательно, поле адреса в таблице не используется.

Таким образом, получили следующую конструкцию строки таблицы компилятора для простой переменной:

обязательные		могут отсутствовать		
идентификатор переменной	тип	признак инициализации	значение	адрес в оттранслированной программе

Описание соответствующей структуры в программе имеет вид

```
struct DataIdent
  {
  TypeObject t;       // для переменной t=ObjVar
  LEX id;             // идентификатор переменной
  int DataType;       // тип значения
   ...                // необязательные данные
  };
```

6.2.2 Константы

Константы в программе можно разделить на два типа:

1) именованные константы, которые описываются в программе с помощью специальных операторов, например, в программе на языке Паскаль, можно описать константы с помощью специального оператора *const*

```
Const Max=10000;
      eps=1e-5;
```

2) константы, которые представлены в программе своим изображением, например, в программе на языке Си можно написать

```
char t[100]="This program must be run under Win32\n";
                             // это строковая константа
float a=3.14156*r*2;         // вещественная и целая константа
```

Любая константа полностью представима следующими данными:

обязательные			может отсутствовать
идентификатор константы	тип	изображение константы	адрес в оттранслированной программе

Именованные константы лучше занести в таблицу переменных со специальным признаком константы. В этом случае при обработке любого идентификатора в тексте программы поиск всегда будет осуществляться в единой таблице, а не сначала в таблице переменных, а потом в таблице констант.

Если представленная свои изображением константа может являться непосредственным операндом ассемблерной команды, то заносить такую константу в таблицу нет смысла. Если же константу представить непосредственным операндом невозможно, ее нужно занести в таблицу и в дальнейшем при генерации ассемблерной программы придется работать с адресами или сгенерированными именами этих команд. Константы такого типа являются длинными и занимают много места, поэтому нельзя в таблице резервировать поле фиксированной длины для значения константы. В этом случае придется воспользоваться динамическим выделением памяти.

Обобщая все вышесказанное, приходим к следующей структуре данных для представления одной переменной или одной константы:

```
struct DataIdent
  {
  TypeObject t;      // для переменной и константы t=ObjVar
  LEX id;            // идентификатор переменной или константы
  int DataType;      // тип значения
  int FlagConst;     // признак константы
  char * Data;       // ссылка на значение константы или NULL
         // необязательные данные:
  int FlagInit;      // Флаг начальной инициализации
  char * Addr;       // адрес в оттранслированной программе
  };
```

6.2.3 Массивы

Описание массивов в большинстве языков программирования построено на задании в момент описания числа измерений и границ по каждому измерению. Эта информация требуется, во–первых, чтобы выделить место в памяти для массива, а во–вторых, для вычисления смещения индексированной переменной.

обязательные					
идентификатор массива	тип элемента	размерность массива	измерение 1	измерение 2	...

могут отсутствовать		
признак инициализации	значение	адрес в оттранслированной программе

В зависимости от стандарта описания массива в языке программирования информация по каждому измерению представляет собой либо два числа — верхнюю и нижнюю границу (как, например, в языке Паскаль), либо только одно число — верхнюю границу измерения или количество элементов по данному измерению (как, например, в языке Си), т.к. информация о нижних границах является избыточной, если в транслируемом языке программирования она всегда является фиксированной. В некоторых языках программирования, например, в языке Java, границы массива объявляются динамически, как указано в следующих примерах:

```
int MyArray1[]   = new int[100];
int MyArray2[][];         MyArray2=new int[100];
int MyArray3[][] = new int[20][];
MyArray3[4]=new int[10];  MyArray3[5]=new int[20];
```

При трансляции таких языков программирования в таблице компилятора хранится только число измерений массива, а информация о каждом измерении переносится на уровень выполнения программы.

Описание соответствующей структуры в программе имеет вид

```
struct DataArray
  {
  TypeObject t;    // для массива t=ObjArray
  LEX id;          // идентификатор массива
  int DataType;    // тип значения
  int N;           // размерность массива
  int lg[MAX_N];   // нижние границы измерений
  int hg[MAX_N];   // верхние границы измерений
   ...             // необязательные данные
  };
```

6.2.4 Структуры

Поскольку элементом записи может быть любой объект, то необходимо реализовать гибридную структуру таблиц в виде дерева независимо от того, допускает язык программирования блочную структуру или нет. Элемент дерева гибридной таблицы содержит ссылки на левого и правого потомков, а также структуру ”union” по тем типам данных, которые описывают один элемент таблицы для простых переменных или массивов.

Реализацию описания структуры рассмотрим в конце данного параграфа.

6.2.5 Функции и процедуры

Каждая функция любого языка программирования имеет имя, может возвращать или не возвращать значения, может иметь или не иметь параметры. Все эти характеристики функции должны быть отражены в таблице компилятора. Описание каждого параметра функции должно содержать тип параметра. Кроме того, в таблице функций может присутствовать имя каждого параметра. Однако, для простоты реализации семантических подпрограмм это имя лучше поместить в обычную таблицу данных с признаком того, что имя является параметром. Лучший вариант реализации таблицы функций и всех связанных с ней данных основан на древовидной структуре таблицы. В такой таблице вершина, соответствующая функции, имеет правые потомки при условии, что она содержит локальные данные или параметры. Первые в списке правые потомки — ее параметры, далее список состоит из локальных данных функции.

Существует два способа передачи параметров — по ссылке и по значению. Такое разделение параметров на типы исторически сложилось при изучении языков программирования для характеристики тех данных, которые передаются функции. При передаче параметров по значению в стек записываются сами значения параметров, внутри функции эти параметры могут только обрабатываться и при выходе из функции фактические параметры остаются без изменения. При передаче параметров по наименованию в стек записывается адрес параметра и, следовательно, все происходящее со значением параметра отражается на значении аргумента. На самом деле, с точки зрения разработчика компилятора, все параметры передаются по значению, только значения эти бывают разные — данные или их адреса. Это замечание оказывает существенное влияние на способ хранения информации о том, по ссылке или по значению переданы параметры. В простейшем варианте реализации можно для каждого параметра хранить в таблице специальный признак. Более общий вариант основан на представлении типа каждого параметра, в том числе и адресного.

Таким образом, для функции в таблице компилятора необходимо хранить следующую информацию:

обязательные					
идентификатор функции	тип возвращаемого значения	число параметров	тип параметра 1	тип параметра 2	...

могут отсутствовать	
объем локальных данных	адрес в оттранслированной программе

Если таблица реализована в виде древовидной структуры, то параметры занимают отведенное им место в дереве. Следовательно, в вершине, соответствующей функции, типы параметров хранить не имеет смысла. Достаточно иметь информацию о количестве параметров. Тогда информационная часть структуры для функции имеет вид:

```
struct DataFunc
  {
  TypeObject t;     // для функции t=ObjFunct
  LEX id;           // идентификатор функции
```

```
  int DataType;    // тип возвращаемого функцией значения
  int Param;       // количество параметров
  };
```

6.2.6 Метки

Метки ипользуются в программе в двух случаях:

а) метка может помечать некоторый оператор программы; такое появление метки в программе может считаться ее объявлением;

б) метка может использоваться в операторе *goto*, определяя тем самым переход вперед или назад по тексту программы.

Нельзя выполнить переход на метку, которой в программе нет, но можно поставить метку, на которую нет перехода. Поэтому появление метки, помечающей оператор, должно вызвать занесение этой метки в таблицу с признаком описания. Появление метки в операторе *goto* должно вызывать занесение метки в таблицу только в том случае, если данная метка в таблице отсутствует. Причем в последнем случае занесение метки должно сопровождаться установкой флага отсутствия описания.

Информация о метке состоит только из двух обязательных полей:

обязательные		могут отсутствовать
идентификатор метки	признак описания	адрес в оттранслированной программе

Ей соответствует описание обязательных полей

```
struct DataLabel
  {
  TypeObject t;  // для метки t=ObjLabel
  LEX id;       // идентификатор метки
  int FlagDef;  // Флаг описания
  };
```

6.2.7 Типы

Типы в программе могут иметь или не иметь имя, но всегда имеют реализацию в виде некоторой допустимой в языке программирования конструкции из других типов данных. Это означает, что типы представимы в программе компилятора точно так же, как и соответствующие этим типам данные. Единственное отличие в представлении заключается в том, что тип должен иметь в таблице специально установленный признак типа:

```
struct TDataType
  {
  TypeObject t;  // для типа значение определяется конструкцией типа:
                 // t=ObjTypeVar, t=ObjTypeArray, t=ObjTypeStruct
  LEX id;        // идентификатор типа
```

```
  int FlagType;  // признак типа
   ...           // позиции, соответствующие данному типу
  };
```

Для простоты понимания структуры данных в таблице мы ввели отдельное обозначение для типа объекта (переменная, массив, структура и т.п.) и для типа данных (int, char, float и т.п.), которые этот объект обозначает. Можно всю эту информацию хранить в одной переменной в виде шкалы флагов–признаков, используя различные разряды для хранения разной информации о структуре элемента таблицы. Тогда признак типа и объекта — это одно из значений переменной DataType из описания переменной. Более того, собирая вместе все необходимые к представлению в таблице данные для переменных, констант, массивов, структур, функций, меток и типов, получаем одну общую конструкцию

```
struct Node
  {
         // данные, общие для всех типов объектов
  LEX id;            // идентификатор объекта
  int DataType;      // тип значения ( в том числе и признак типа)
         // обязательные данные для некоторых типов объектов
  int FlagConst;     // признак константы
  char * Data;       // ссылка на значение константы или NULL
  int Param;         // количество параметров функции
  int N;             // размерность массива
  int lg[MAX_N];     // нижние границы измерений  массива
  int hg[MAX_N];     // верхние границы измерений  массива
  int FlagDef;       // Флаг описания  метки
         // необязательные данные:
  int FlagInit;      // Флаг начальной инициализации
  char * Addr;       // адрес в оттранслированной программе
  };
```

Для сокращения объемов памяти можно некоторые из данных, уникальных для объектов определенного вида, разместить в той же области памяти, что и уникальные данные для объектов другого вида. Для этого имеет смысл воспользоваться конструкцией *union*. Выделение перекрываемых данных в описании таблицы зависит от транслируемого языка программирования и оставляется в качестве упражнения.

6.2.8 Программирование таблицы компилятора

Если в качестве структуры таблицы выбраны массивы, то работа с такой таблицей является тривиальной и здесь рассматриваться не будет. Рассмотрим наиболее часто встречающийся вариант реализации таблицы в виде дерева. На практике обычно выбирают один из двух способов реализации дерева:

1) дерево представлено массивом;

2) дерево — это динамическая структура.

Первый вариант проще в реализации, но накладывает ограничения на максимальный размер дерева и, кроме того, требует заранее определить размер массива, в котором будет храниться дерево. Второй вариант сложнее в реализации и требует тщательной отладки при работе с адресами, но свободен от указанных ограничений. Следует, однако, отметить, что использование динамической структуры потребует выполнения дорогостоящих операций выделения памяти.

Оба варианта программирования дерева основаны на реализации тех функций, которые необходимы при формировании таблицы и поиска в ней информации:

1) void SetLeft (...) — записать новые данные левым потомком у текущей вершины;

2) void SetRight(...) —записать новые данные правым потомком у текущей вершины;

3) int FindUp(...) — найти идентификатор в дереве от заданной вершины, передвигаясь по дереву только вверх от данной вершины.

Рассмотрим сначала программу соответствующих функций при работе с деревом в форме массива. Для простоты реализации в таблице компилятора будем хранить информацию только о переменных.

```
//**************************************************
// tree_1.hpp    -    представление дерева массивом
//**************************************************
#include "defs.hpp"

#define  MAXK  100    // максимальное число вершин дерева
#define  EMPTY -1     // признак пустой ссылки

struct Node           // информация об одной переменной
  {
  TypeLex id;         // идентификатор переменной
  int DataType;       // тип значения (int, float,...)
  };

class Tree
  {
  Node n[MAXK];       // данные таблицы
  int Up[MAXK], Left[MAXK], Right[MAXK];
      // родитель, левый и правый потомок
  public:
  int Root;    // корень дерева
  int Next;    // следующий заполняемый элемент в массиве
  Tree();
  void SetLeft (int From, Node * Data);
  void SetRight(int From, Node * Data);
  int  FindUp(int From, TypeLex id);
  };


//**************************************************
// tree_1.cpp    -    представление дерева массивом
//**************************************************
#include "defs.hpp"
```

```
#include "tree_1.hpp"
#define max(a,b) a<b? b : a

Tree::Tree (void)
// конструктор создает корень дерева в 0-ом узле
{
Next=0; Root=0;
Up[Next]=EMPTY;  Left[Next]=EMPTY;  Right[Next]=EMPTY;
   // установили вершину  Root
memcpy(&n[Next],&("------"),MAX_LEX);  // в Root нет данных
Next++;
}

void Tree::SetLeft (int From, Node * Data)
// создать левого потомка от вершины From
{
Up[Next]=From;  Left[Next]=EMPTY;  Right[Next]=EMPTY;
           // установили связи в новой вершине
memcpy(&n[Next], Data, sizeof(Node));
           // записали информацию в новую вершину
if(From!=EMPTY) Left[From]=Next;
           // связали From с новой вершиной
Next++;
}

void Tree::SetRight(int From, Node * Data)
// создать правого потомка от вершины From
{
Up[Next]=From;   Left[Next]=EMPTY;   Right[Next]=EMPTY;
           // установили связи в новой вершине
memcpy(&n[Next], Data, sizeof(Node));
           // записали информацию в новую вершину
if(From!=EMPTY) Right[From]=Next;
           // связали From с новой вершиной
Next++;
}

int  Tree::FindUp(int From, TypeLex id)
// поиск данных в дереве до его корня вверх по связям
{
int i=From;                  // текущая вершина поиска
while( (i!=EMPTY) &&
       (memcmp(id, n[i].id, max(strlen(n[i].id),strlen(id)))!=0) )
     i=Up[i];                // поднимаемся наверх по связям
if (i==EMPTY) return EMPTY; else return i;
}
```

Как уже было отмечено, реализация статических структур данных обладает большим выстродействием, но накладывает ограничения на размер хранимых данных.

Остановимся на особенностях реализации древовидной таблицы компилятора в виде динамической структуры. Рассмотренные в приведенной выше программе функции $SetLeft(...)$ и $SetRight(...)$ получили здесь более прозрачную и простую реализацию, основанную на использовании нового конструктора

```
Tree::Tree(Tree * l, Tree * r, Tree * u, Node * Data).
```

Кроме того, будем строить таблицу в предположении, что язык программирования допускает использование структур. Тогда функция $FindUp(...)$ должна быть реализована в двух вариантах: поиск нужных данных от текущей вершины *this* и от заданной вершины $From$. К множеству функций работы с древовидной структурой здесь следует добавить еще очень важную функцию

```
          Tree*  FindRightLeft  (...).
```

Эта функция выполняет поиск данных только по соседям одного уровня вложенности (вспомним, что при формировании таблицы в виде дерева мы решили считать левые потомки соседями, а правые — данными следующего уровня вложенности). Эта функция, как ясно из анализа представления таблицы, необходима при поиске подструктур заданной структуры. Все подструктуры следующего уровня иерархи от данной структуры A— это левые соседи правого потомка A.

```
//*************************************************
// tree_2.hpp
//       представление дерева динамической структурой
//*************************************************
#include "defs.hpp"

struct Node
  {
  TypeLex id;        // идентификатор переменной
  int DataType;      // тип значения
  };

class Tree
  {
  Node * n;          // данные таблицы
  Tree * Up, * Left, * Right;
                     // родитель, левый и правый потомок
  public:
  Tree(Tree * l, Tree * r, Tree * u, Node * Data);
  Tree();
  ~Tree();
  void SetLeft (Node * Data);
  void SetRight(Node * Data);
  Tree *  FindUp  (Tree * From, TypeLex id);
  Tree *  FindUp  (TypeLex id);
  Tree *  FindRightLeft  (Tree * From, TypeLex id);
  Tree *  FindRightLeft  (TypeLex id);
  void Print(void);
```

```
  };

//**************************************************
// tree_2.cpp
//        представление дерева динамической структурой
//**************************************************
#include "defs.hpp"
#include "tree_2.hpp"
#define max(a,b) a<b? b : a

Tree::Tree (Tree  * l, Tree * r, Tree * u, Node * d)
// конструктор создает узел с заданными связями и данными
{
n= new Node();
Up=u;  Left=l;  Right=r;           // установили ссылки
memcpy(n, d, sizeof(Node));        // установили данные
}

Tree::Tree (void)
// конструктор создает новый узел с пустыми связями и данными
{
n= new Node();
Up=NULL;  Left=NULL;  Right=NULL;
memcpy(n,&("------"), sizeof(Node));
}

void Tree::SetLeft (Node * Data)
// создать левого потомка от текущей вершины this
{
Tree * a= new Tree(NULL,NULL,this,Data);   // новая вершина
Left=a;                    // связали this с новой вершиной
}

void Tree::SetRight(Node * Data)
// создать правого потомка от текущей вершины this
{
Tree * a= new Tree(NULL,NULL,this,Data);   // новая вершина
Right=a;                   // связали this с новой вершиной
}

Tree *  Tree::FindUp(TypeLex id)
// поиск данных в дереве, начиная от текущей вершины this
// до его корня вверх по связям
{
return FindUp(this, id);
}

Tree *  Tree::FindUp(Tree * From, TypeLex id)
// поиск данных в дереве от заданной вершины From
// до его корня вверх по связям
```

```
{
Tree * i=From;          // текущая вершина поиска
while( (i!=NULL) &&
       (memcmp(id, i->n->id, max(strlen(i->n->id),strlen(id)))!=0) )
     i=i->Up;           // поднимаемся наверх по связям
return i;
}

Tree *  Tree::FindRightLeft(TypeLex id)
// поиск прямых потомков текущей вершины this
{
return FindRightLeft(this, id);
}

Tree *  Tree::FindRightLeft(Tree * From, TypeLex id)
// поиск прямых потомков заданной вершины From
{
Tree * i=From->Right;         // текущая вершина поиска
while( (i!=NULL) &&
       (memcmp(id, i->n->id, max(strlen(i->n->id),strlen(id)))!=0) )
     i==i->Left;
            // обходим только соседей по левым связям
return i;
}


void Tree::Print (void)
// отладочная программа печати дерева
{
printf("Вершина с данными %s ----->",  n->id );
if (Left !=NULL) printf("     слева данные  %s",  Left->n->id );
if (Right!=NULL) printf("     справа данные %s", Right->n->id );
printf("\n");
if (Left!=NULL) Left->Print();
if (Right!=NULL) Right->Print();
}
```

В качестве замечания следует только сделать следующее предупреждение. Известно, что память выделяется блоками, минимальный размер которого не может быть меньше параграфа. Поэтому стремление выделять память очень мелкими фрагментами может привести к неоправданному расходу памяти.

6.3 Семантические подпрограммы и их вызовы

Любые синтаксические диаграммы, связанные с выражениями, должны вычислять тип соответствующего выражения. Использование типа в процессе компиляции преследует следующие цели:

а) для контроля допустимости операции над выражениями или использования выражения определенного типа в некотором операторе;

б) для формирования команд преобразования значения из одного типа в другой тип при выполнении операций над данными разных типов.

Возвращаемый тип выражения хорошо использовать в качестве формального параметра функции, соответствующей синтаксической диаграмме. Рассмотрим два типа синтаксических диаграмм, связанных с выражением — элементарное выражение и выражение, содержащее операции над данными. Пусть в элементарном выражении допускаются идентификаторы простых переменных, константы и выражения в скобках. Пусть t — параметр, представляющий собой тип возвращаемого значения (например, $integer$, $float$,...). Тогда синтаксическая диаграмма для элементарного выражения E со встроенными подпрограммами семантического контроля типов имеет вид, представленный на рис. 6.3.

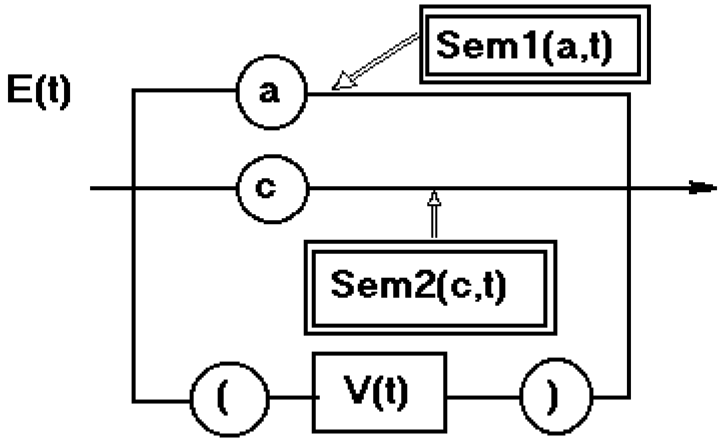


Рис. 6.3: Семантический контроль элементарного выражения

Очевидно, что если элементарное выражение — это выражение в скобках, то тип элементарного выражения совпадает с типом выражения в скобках, поэтому совпадают и параметры функций $E(t)$ и $V(t)$.

Семантические подпрограммы $Sem1(a,t)$ и $Sem2(c,t)$ в качестве первого параметра имеют отсканированную лексему, а в качестве второго — возвращаемый тип. Семантическая подпрограмма $Sem1(a,t)$ работает по–разному в зависимости от свойств транслируемого языка программирования. В языке без умолчания (Си, Паскаль) необходимо найти объект в таблице идентификаторов, определить тип этого объекта и вернуть этот тип в качестве результата t. При отсутствии переменной a в таблице выдается сообщение о семантической ошибке и возвращается "неопределенный тип". Следует отметить, что семантическая ошибка не оказывает влияния на блок нейтрализации синтаксиса и не требует никаких дополнительных действий для того, чтобы продолжить синтаксический анализ. Однако, для предотвращения дополнительных семантических ошибок следует предусмотреть, чтобы семантическая программа контроля допустимости данных с "неопределенным типом " всегда выполняла действия корректно.

В языке с умолчанием семантическая функция $Sem1(a,t)$ при отсутствии объекта в таблице заносит его туда и присваивает тип в соответствии с соглашением о типах данного языка программирования (обычно тип определяется по умолчанию по первой букве идентификатора).

Семантическая функция $Sem2(c,t)$ помещает константу в таблицу констант, если это требуется и при условии реализации этой таблицы, а также возвращает тип константы. В дальнейшем таблица констант будет оттранслирована в объектный код в блоке инициализированных данных. Функция $Sem2(c,t)$ может иметь и дополнительный параметр — тип лексемы, который возвращает сканер. В этом случае существенно упрощается алгоритм определения типа t.

Если в выражении используются элементы структур, то следует проверить последовательность разделенных точкой идентификаторов на соответствие описанию структуры. Для элемента массива нужно проверить размерность соответствующего массива, а также тип каждого индекса. Как правило, в качестве индекса может выступать любое выражение, поэтому проверка типа каждого индекса заключается в проверке приведения типа выражения к типу, который должен являться индексом соответствующего измерения. Например, в языке Си таким типом должно быть целое число.

Рассмотрим теперь синтаксическую диаграмму для выражения, в котором используются операции над данными (см. рис. 6.4).

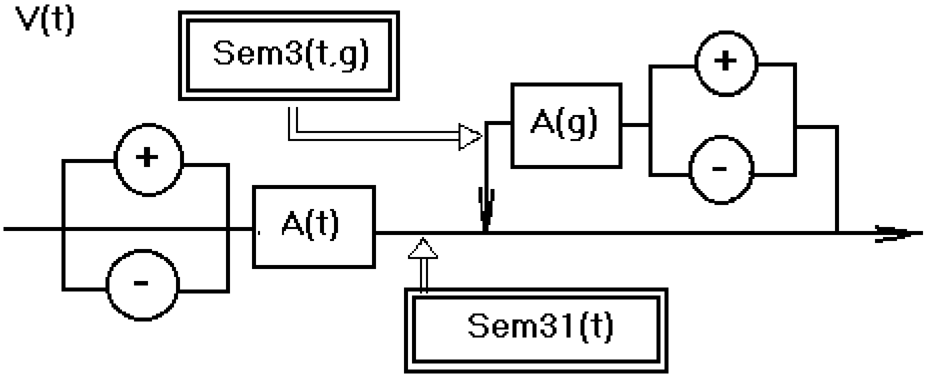


Рис. 6.4: Семантический контроль в выражении

Семантическая подпрограмма $Sem31(t)$ проверяет допустимость выполнения унарной арифметической операции "+" или "-" над данными типа t, а подпрограмма $Sem3(t,g)$ вычисляет тип результата операции над операндами t и g в соответствии с таблицей приведений. Для операций сложения и вычитания и простейших арифметических типов данных таблица приведений может иметь вид:

операнд 1	операнд 2	результат
integer	integer	integer
float	integer	float
integer	float	float
float	float	float
неопределенный тип	неопределенный тип float, integer	неопределенный тип
неопределенный тип float, integer	неопределенный тип	неопределенный тип

Еще раз отметим, что операция, один из операндов которой имеет "неопределенный тип" всегда возвращает в качестве значения также "неопределенный тип".

Во многих языках программирования в одной диаграмме могут выполняться многие операции (например, все аддитивные или все мультипликативные), тип результата которых существенно зависит от вида самой операции. Например, операция "/" над целыми данными в Си вернет в качестве значения результат типа $float$, а операция "над целыми — результат типа int. Поэтому $Sem3$ в общем случае должна

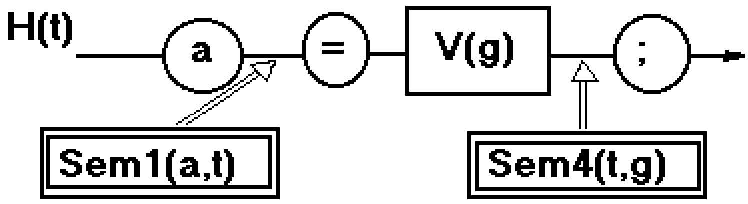


Рис. 6.5: Семантический контроль в операторе присваивания

учитывать не только типы операндов, но и вид операции. Следовательно, необходимо либо внести дополнительный параметр в список параметров функции $Sem3$ — знак операции, либо использовать уникальные семантические подпрограммы для каждой операции.

Отметим также, что при генерации объектного кода выполнение большинства операций над данными разных типов вызывает генерацию команд обращения к функции преобразования данных из одного типа в другой. Эту генерацию можно встроить непосредственно в $Sem3$.

Перейдем теперь к анализу контекстных условий в более сложных конструкциях языка, в частности к анализу операторов вызова функции, циклов, условных операторов и т.п. В языках программирования реализуется один из двух подходов к типам значений, связанных с операторами:

а) любой оператор не возвращает значения;

б) каждый оператор возвращает значение, в том числе и значение типа *void*.

Реализация такого контроля предполагает либо отсутствие либо наличие параметров в соответствующей функции синтаксической диаграммы. Например, простейший оператор присваивания языка Си имеет вид, представленный на рис. 6.5.

Семантический контроль вызовов функций должен обеспечивать либо проверку совпадения типов формальных и фактических параметров, либо контролировать приводимость типа фактического параметра к типу формального. Необходимо также проверить, чтобы формальные и фактические параметры совпадали по количеству. Семантический контроль операторов различного вида, в которых используются выражения, должен обеспечивать проверку типа выражения с тем, чтобы оно было приводимо к тому типу, который допускает соответствующая синтаксическая конструкция. Например, в операторе if или $while$ языка Паскаль выражение может быть только типа *boolean*.

6.4 Трансляция описаний

6.4.1 Простые переменные и массивы

Описание данных в различных языках программирования выполняется по–разному. Чаще всего, как, например, в языке Си, указывается тип данных, а затем перечисляются идентификаторы. Такое описание легко реализуется компилятором, т.к. тип данных перед их занесением в таблицу уже известен. Рассмотрим, например, упрощенную структуру оператора описания языка Си. Пусть допускаются типы int, $float$ и пользовательский тип данных, который обозначается идентификатором. Можно описать простые переменные и массивы любой размерности, причем в качестве размера по каждому измерению можно указывать только константы. Соответствующая синтаксическая диаграмма представлена на рис. 6.6.

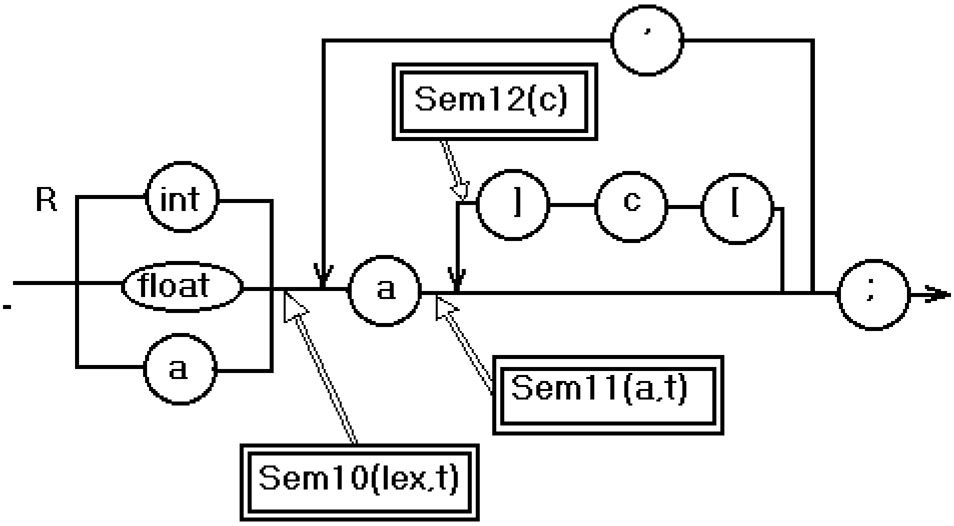


Рис. 6.6: Семантика оператора описания переменных и массивов языка Си

Рассмотрим семантическую обработку этого описания. Сначала при сканировании первой лексемы необходимо определить семантический тип для занесения его в таблицу в поле $DataType$ для каждой переменной или массива. Для этой цели служит семантическая подпрограмма $Sem10(lex, t)$, которая по изображению типа lex определяет семантический тип t. Далее при сканировании каждого очередного идентификатора a его необходимо с типом t занести в таблицу. Для этого используется семантическая подпрограмма $Sem11(a, t)$. Поскольку в языке допускается как описание простых переменных, так и массивов, то при занесении каждого идентификатора a в таблицу сначала следует установить число измерений, равное нулю. В дальнейшем, если появится описание размерности, нужно будет занести в таблицу компилятора информацию о каждом измерении объекта a, а также откорректировать число измерений. Для этой цели используется семантическая подпрограмма $Sem12(c)$, которая увеличивает на единицу число измерений для текущего данного и запоминает значение размерности, полученное преобразованием числа c из символьного представления в int.

В языке Паскаль принцип описания совершенно иной: сначала указывается список идентификаторов, а затем после знака ":" ставится описание типа. Такая структура оператора описания приводит к необходимости двойной обработки каждого элемента таблицы: сначала в таблицу заносятся объекты без указания их типа, а затем для всех вновь занесенных объектов заполняется поле типа и размерность.

Один из вариантов такой обработки семантики представлен на рис. 6.7. Семантическая подпрограмма $Sem13()$ возвращает указатель на текущую позицию в таблице компилятора. Подпрограмма $Sem14(a)$ заносит объект a в таблицу без указания его типа. Подпрограмма $Sem12(c, c)$ практически совпадает с аналогичной программой рис. 6.6, за тем исключением, что измерение в Паскале характеризуется двумя числами — верхней и нижней границей. После определения типа объекта семантической подпрограммой $Sem10(lex, t)$ можно занести этот тип в таблицу для всех объектов, которые были занесены в процессе трансляции данного описания. Эту задачу выполняет семантическая подпрограмма $Sem15(n, t)$, где n — указатель на последний перед началом занесения новых объектов элемент таблицы.

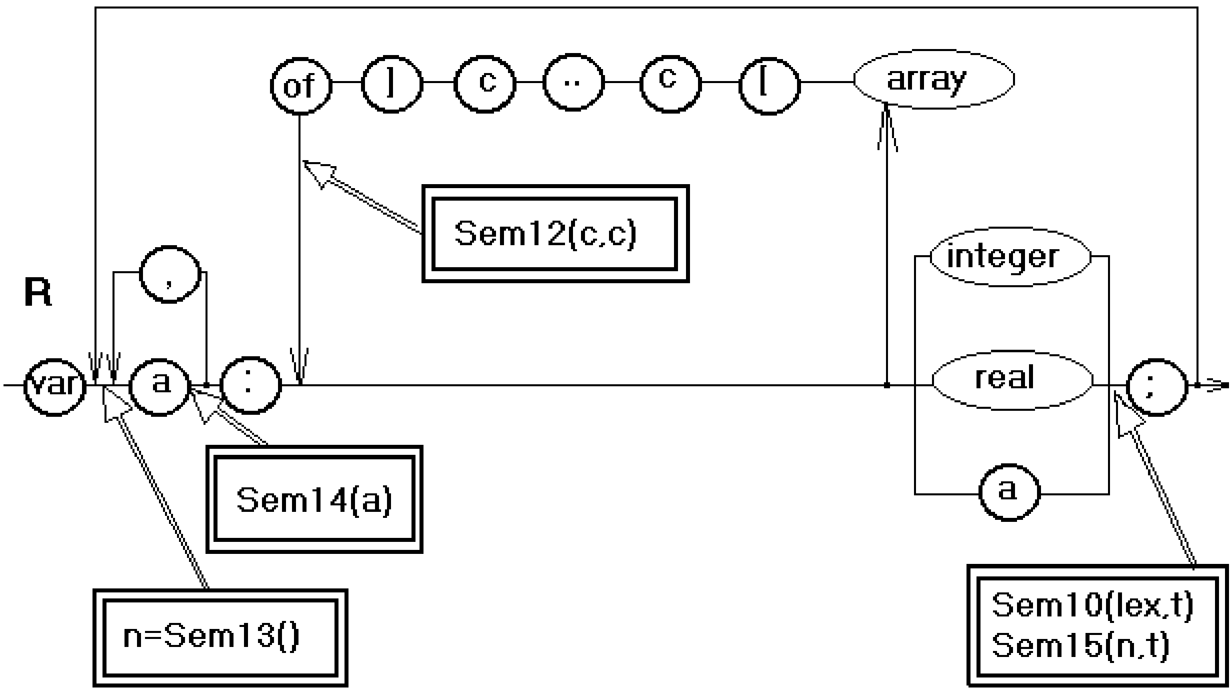


Рис. 6.7: Семантика оператора описания переменных и массивов языка Паскаль

6.4.2 Функции

Как уже отмечалась в 6.1, наличие блочной структуры программы требует древовидной структуры таблицы компилятора. Рассмотрим описание функции, представленное на рис. 6.8. Пусть функция имеет простейшие параметры и возвращает значение. Как параметры, так и возвращаемое функцией значение имеет тип *int*, *float* или пользовательский тип данных, который представлен идентификатором.

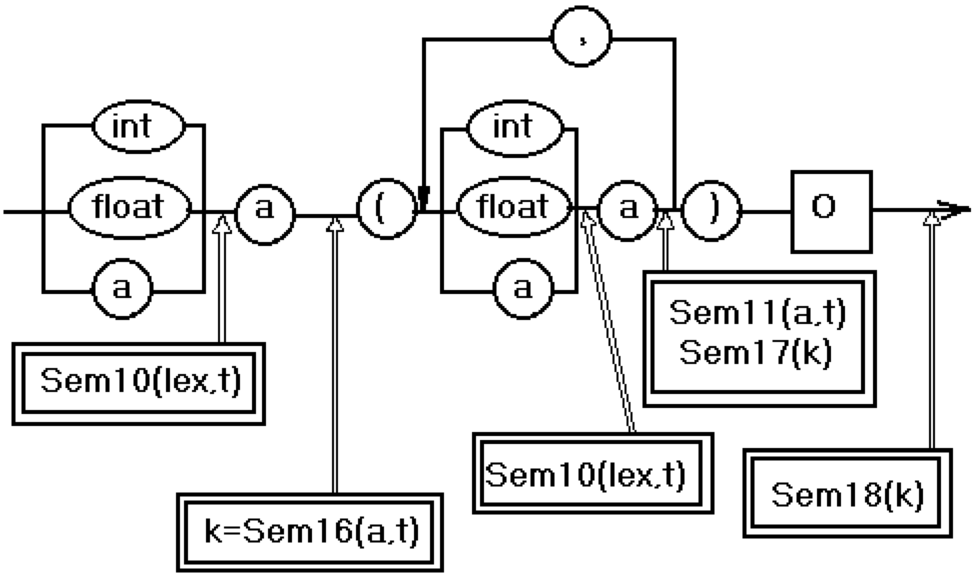


Рис. 6.8: Семантика описания функции языка Си

Все параметры и локальные данные должны быть вставлены в таблицу в качестве правых потомков вершины, которая является именем функции. Для этой цели используется семантическая подпрограмма $Sem16(a, t)$, которая заносит в таблицу идентификатор a в качестве имени функции, возвращающей тип t. После этого $Sem16(a, t)$ создает "пустого" правого потомка вершины a, устанавливает на него указатель текущего объекта таблицы, а затем возвращает в качестве результата указатель на созданную вершину a. Это значение указателя будет использовано в конце обработки функции семантической подпрограммой $Sem18(k)$, которая восстановит текущее значение указателя таблицы по значению k.

Перейдем теперь к реализации параметров функции. Когда имя функции заносится таблицу, число параметров этой функции еще не известно. Поэтому при формировании элемента таблицы для функции количество параметров этой функции устанавливается в ноль. Затем при появлении каждого нового параметра семантическая подпрограмма $Sem17()$ увеличивает на единицу число параметров функции. Параметры функции могут как иметь, так и не иметь в таблице специального признака "параметр т.к. эта информация легко восстанавливается по количеству параметров у функции. Однако, для простоты обработки параметров внутри тела функции лучше такой признак устанавливать. Эту задачу также может решать подпрограмма $Sem17()$.

6.4.3 Структуры

Наличие структур в языке программирования приводит к обязательному использованию древовидной конструкции таблицы компилятора. Рассмотрим пример описания структур, представленный на рис. 6.9, построенный с использованием уже рассмотренной ранее конструкции описания переменных R (см. рис. 6.6).

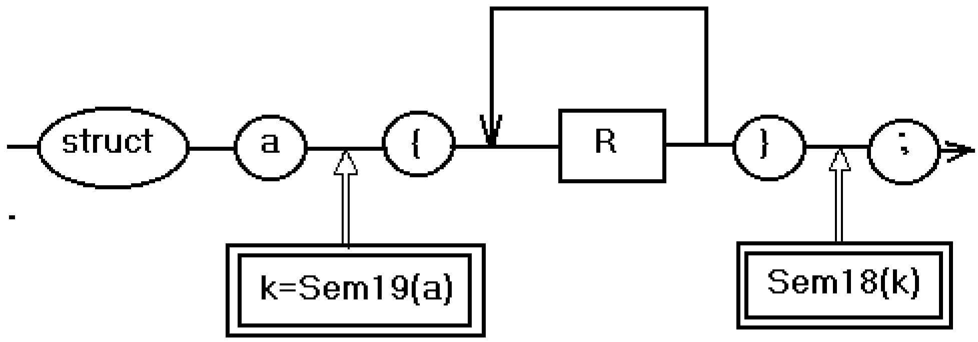


Рис. 6.9: Семантика оператора описания структур языка Си

Для занесения структуры в таблицу достаточно воспользоваться двумя семантическими подпрограммами. Первая из них $Sem19(a)$ является аналогом уже рассмотренной $Sem16(a, t)$. Она заносит в дерево очередной элемент a как тип, создает правую ссылку, устанавливает текущее положение указателя на созданный правый элемент, возвращает указатель на ту вершину, в которой находится созданный элемент a — имя структурного типа. После этого все внутренние элементы структуры a в диаграмме R будут заноситься по обычным правилам в дерево и, следовательно, окажутся правыми потомками вершины a. После завершения описания структуры необходимо восстановить текущее положение указателя на вершину a с тем, чтобы все последующие описания появились на том же уровне иерархии, на котором находится a. Эту задачу выполняет семантическая подпрограмма $Sem18(k)$.

6.5 Контрольные вопросы к разделу 4

1. Какая информация хранится в таблице компилятора для простых переменных?

2. Когда заносится информация в таблицу компилятора?

3. Зачем и когда используется древовидная структура таблицы компилятора?

4. Поясните смысл семантических подпрограмм при реализации элементарного выражения.

5. Напишите программу формирования таблицы в виде дерева.

6. Если в языке программирования допускается описание пользовательских типов, то какая дополнительная информация хранится в таблице компилятора?

7. Информация о каких объектах программы хранится в таблицах компилятора?

8. Какая информация хранится в таблице компилятора для массивов?

9. Поясните смысл семантических подпрограмм при реализации оператора присваивания.

10. Какие контекстные условия проверяются для вызова функции?

11. Что такое таблица приведений?

12. Чем различаются семантические подпрограммы в языке с умолчанием и в языке без умолчания?

13. Как определить тип выражения.

14. Поясните смысл понятия "неопределенный тип".

15. Чем различаются таблицы для хранения информации о массивах при трансляции языков Си и Паскаль?

16. Какие контекстные условия проверяются для условных и циклических операторов?

17. Поясните смысл семантических подпрограмм при реализации операторов описания.

18. Какая семантическая информация связана с метками?

19. Приведите пример оператора, при трансляции которого не требуется семантический контроль.

20. Какая информация хранится в таблице компилятора для функций?

6.6 Тесты для самоконтроля к разделу 4

1. Какое из перечисленных ниже толкований контекстных условий наиболее полно отражает их смысл?

Варианты ответов.

а) Это правила, по которым вычисляется значение переменных в процессе интерпретации программы.

б) Это контекст, в пределах которого может находиться сканируемый идентификатор.

в) Это правила использования идентификаторов и констант, которые нельзя описать на уровне КС–грамматик.

г) Контекстные условия определяют структуру таблицы компилятора.

д) Контекстные условия задаются правилами приведения типов.

Правильный ответ: в.

2. Какие из перечисленных ниже условий приводят к эффективному использованию древовидной структуры таблицы компилятора? Укажите наиболее полный ответ.

Варианты ответов.

а) Наличие блочной структуры программы.

б) Наличие операторов описания данных разных типов.

в) Возможность описывать пользовательские типы данных.

г) Возможность описания функций или процедур .

Правильный ответ: а.

3. Очень простой интерпретируемый язык программирования предназначен только для линейного вычислительного процесса и не поддерживает блочную структуру программы. Этот язык допускает использование следующих типов данных: целые, целые короткие и целые длинные (соответственно 4 байта, 2 байта и 8 байтов); вещественные и вещественные длинные (соответственно 4 байта и 8 байтов). Длина идентификаторов не превышает 3 символов. Что Вы можете сказать о следующем описании элемента таблицы в виде динамической древовидной структуры

```
struct TypeTree              // информация в вершине дерева
  {
  char *  id;                // идентификатор объекта
  DATA_TYPE DataType;        // тип значения
  void * Data;               // ссылка на значение переменной
  TypeTree  * Up, * Left, * Right;
  };
```

Варианты ответов.

а) Описание совершенно правильное и будет эффективно работать.

б) Хранить тип значения нерационально, т.к. ссылка $Data$ уже указывает на правильные данные.

в) Динамическое выделение памяти для хранения id и $Data$ нерационально, т.к. будет использоваться лишняя память из-за особенностей системы выделения памяти. Следует ограничиться статическимми элементами для указанных данных,

г) Использование динамической структуры для дерева возможно всегда в силу ее универсальности, но в данном случае можно было бы ограничиться и статической (табличной) структурой.

д) В данном случае надо устранить все динамические конструкции.

Правильный ответ: д.

4. Что представляет собой процесс приведения типов? Укажите наиболее правильный и точный ответ.

Варианты ответов.

а) Это вычисление типа результата операции по типам операндов этой операции.

б) Это вычисление значения результата операции по значениям операндов.

в) Это контроль правильности использования операндов в некотором контексте.

г) Это преобразование выражения в правой части оператора присваивания к типу переменной в левой части этого оператора.

д) Это процесс поиска переменной в таблице.

Правильный ответ: а.

5. Какую информацию для процедур и функций необходимо хранить в одном элементе таблицы?

Варианты ответов:
а) имя функции, ее тип, число параметров, типы параметров;
б) имя функции, ее тип, число параметров, идентификаторы и типы параметров;
в) имя функции, ее тип, число параметров;
г) имя функции, ее тип;
д) имя функции.

Правильный ответ: в.

6.7 Упражнения к разделу

6.7.1 Задание

Цель данного задания – реализация семантического контроля в программе транслятора. Чтобы выполнить задание, Вам рекомендуется проделать следующие действия.

1. Выделить типы контекстных условий, проверка которых необходима в языке Вашего задания.

2. Выбрать структуру таблиц компилятора. При этом на структуру таблицы самое существенное влияние должны оказать следующие свойства языка программирования, соответствующего Вашему заданию:

- наличие блочной структуры программы;
- наличие структур (или записей);
- наличие функций и процедур;
- возможность объявления типов данных;
- встроенные типы данных языка программирования (многомерные и одномерные массивы, множества, именованные константы и т.п.).

3. Определить перечень семантических подпрограмм обработки спроектированных Вами таблиц.

4. Выполнить разметку синтаксических диаграмм:

- для каждой синтаксической диаграммы указать точки вставки семантических подпрограмм;
- указать наименование вызываемой подпрограммы.

5. Указать действия, соответствующие начальной инициализации (очистку таблиц, начальную установку указателей и т.п.).

6. В соответствии с разметкой синтаксических диаграмм разработать описание типов данных, реализующие предлагаемые Вами таблицы компилятора.

7. Определить список параметров для каждой семантической подпрограммы. Выделить глобальные данные (если Вы считаете разумным их использование), относящиеся к семантическому уровню реализации компилятора.

8. Внести дополнения в модуль $defs.hpp$, соответствующие новым типам данных компилятора, которые реализуют семантический уровень языка.

9. Внести дополнение в модуль, содержащий определение глобальных типов данных, включив в него данные семантического уровня.

10. Написать семантические подпрограммы. Определить модули *semant.cpp*, *semant.hpp*, включающие соответственно реализацию семантических подпрограмм и их заголовки. Процесс реализации семантических подпрограмм лучше провести

по двухшаговому принципу: сначала пишутся и отлаживаются функции обработки базовой табличной структуры (например, работа с деревом), а затем отлаживаются семантические подпрограммы контроля контекстных условий.

11. Дополнить список ошибок, которые выдает программа $PrintError()$.

12. Разработанные семантические подпрограммы встроить в программу синтаксического анализа, работающую до первой ошибки, который Вы построили, выполняя упражнение к разделу 3.

13. Дополнить файл проекта.

14. Отладить программу, обращая особое внимание на набор тестов. Проверить реакцию Вашего компилятора на синтаксические ошибки, которые Вы уже проверяли при отладке программы синтаксического анализа: вставка семантических подпрограмм не должна повлиять на эту реакцию.

15. Проверить реакцию Вашего компилятора на семантические ошибки разного рода: повторные объявления данных; использование необъявленных данных; несогласование типов данных в одном операторе; невозможность приведения типов данных к требуемому типу и т.п.

6.7.2 Пример выполнения задания

В транслируемом подмножестве языка Java–Script будем проверять следующие контекстные условия:

1) все переменные должны быть описаны в пределах того блока, в котором они встречаются;

2) все функции должны быть описаны;

3) число фактических параметров функций должно совпадать с числом формальных параметров;

4) поскольку в языке Java–Script тип переменной определяется в момент присваивания значения этой переменной, то контроль приведения типов может осуществляться только в процессе интерпретации выражения; следовательно, на данном этапе разработки программы мы не можем выполнить соответствующий контроль;

5) для простоты и наглядности примера будем считать, что объекты в HTML-документе не используются, а, следовательно, и отсутствуют уточненные имена.

Блочная структура программы означает необходимость использования древовидных таблиц. В качестве базовой структуры одного элемента таблицы примем конструкцию, представленную на странице 180, оставив в списке полей только те, которые описывают типы данных транслируемого подмножества языка Java–Script:

```
enum DATA_TYPE {TYPE_UNKNOWN=1, TYPE_INTEGER,
                TYPE_FLOAT,     TYPE_CHAR, TYPE_FUNCT
               };
struct Node
  {
  LEX id;                    // идентификатор объекта
  DATA_TYPE DataType;        // тип значения
  char * Data;               // ссылка на значение или NULL
  int Param;                 // количество параметров функции
  };
```

Рассмотрим перечень семантических подпрограмм, которые потребуется реализовать для контроля контекстных условий:

```
1) Tree * SemInclude(LEX a, DATA_TYPE t) -
```

занесение идентификатора a в таблицу с типом t (при занесении в качестве типа может выступать только значение

```
TYPE_FUNCT
```

— для функций и значение

```
TYPE_UNKNOWN
```

— для переменных); функция возвращает указатель на созданную вершину;

```
2) void SemSetType(Tree Addr, DATA_TYPE t) -
```

установить тип t для той переменной, которая хранится в таблице по адресу $Addr$;

```
3) void SemSetParam(Tree Addr, int n)  -
```

установить число формальных параметров n для той функции, которая хранится в таблице по адресу $Addr$;

```
4) void SemControlParam(Tree Addr, int n)   -
```

проверить, равно ли число формальных параметров значению n для той функции, которая хранится в таблице по адресу $Addr$;

```
5) Tree * SemGetType(LEX a) -
```

найти в таблице переменную с именем a и вернуть ссылку на соответствующий элемент дерева;

```
6) Tree * SemGetFunct(LEX a) -
```

найти в таблице функцию с именем a и вернуть ссылку на соответствующий элемент дерева.

Следует учесть, что при создании таблицы в виде древовидной структуры функция SemGetFunct(LEX a) должна дополнительно создать правую пустую вершину. Тогда для возврата на исходный уровень вложенности потребуется еще одна семантическая подпрограмма SemRest(Tree Addr).

Необходимо также выполнять контроль числа фактических и формальных параметров функций. Для этого достаточно ввести переменную — счетчик количества параметров, который увеличивается на единицу при сканировании каждого нового параметра. Запись и контроль количества параметров выполняют соответственно функции SemSetParam(Tree Addr, int n) и SemControlParam(Tree Addr, int n).

Точки вызовов соответствующих функций в синтаксических диаграммах очевидны и мы их здесь приводить не будем. Достаточно сказать, что функции поиска

```
SemGetType(LEX a)  и SemGetFunct(LEX a)
```

используются при трансляции выражений и вызовов функций, функция

```
SemInclude(LEX a, DATA_TYPE t) -
```

при трансляции оператора описания данных *var* или тела функции *function*, а

```
SemSetParam(Tree Addr, int n)   -
```

при трансляции списка параметров функций.

Реализацию древовидной таблицы Вы можете использовать в любой из приведенных на странице 183 форм — статической или динамической. Следует только отметить, что реализация поиска повторных описаний данных осуществляется исключительно на одном уровне вложенности. Поэтому потребуется дополнительная функция для такого поиска:

```
Tree *  Tree::FindUpOneLevel(Tree * From, TypeLex id)
// Поиск элемента id вверх по дереву от текущей вершины From.
// Поиск осуществляется на одном уровне вложенности по левым связям
{
Tree * i=From;          // текущая вершина поиска
while( (i!=NULL) &&
       ( i->Up->Right != i)
       )
       {
       if (memcmp(id, i->n->id, max(strlen(i->n->id),strlen(id)))==0)
    return i;  // нашли совпадающий идентификатор
       i=i->Up;         // поднимаемся наверх по связям
       }
return NULL;
}
```

Таким образом, структура файла *semant.hpp* имеет вид:

```
// модуль Semant.hpp -- реализация семантических подпрограмм
#ifndef __SEMAN
#define __SEMAN
#include "defs.hpp"

#define  EMPTY -1

enum DATA_TYPE {TYPE_UNKNOWN=1, TYPE_INTEGER,
                TYPE_FLOAT,     TYPE_CHAR, TYPE_FUNCT
               };
struct Node                 // информация в вершине дерева
  {
  TypeLex  id;              // идентификатор объекта
  DATA_TYPE DataType;       // тип значения
  char * Data;              // ссылка на значение или NULL
  int Param;                // количество параметров функции
  };

class Tree                  // элемент семантической таблицы
```

```
  {
  Node * n;                // информация об объекте таблицы
  Tree * Up, * Left, * Right;
           // родитель, левый и правый потомок
  public:
  static Tree * Cur;       // текущий элемент дерева
// ФУНКЦИИ ОБРАБОТКИ БИНАРНОГО ДЕРЕВА
  Tree(Tree * l, Tree * r, Tree * u, Node * Data);
  Tree();
  ~Tree();
  void SetLeft (Node * Data);
  void SetRight(Node * Data);
  Tree *  FindUp  (Tree * From, TypeLex id);
  Tree *  FindUpOneLevel  (Tree * From, TypeLex id);
  Tree *  FindUp  (TypeLex id);
  void Print(void);

// СЕМАНТИЧЕСКИЕ ПОДПРОГРАММЫ
void   SetCur(Tree * a) ;
// установить  текущий узел дерева

Tree * GetCur(void);
// получить значение текущего узла дерева

Tree * SemInclude(TypeLex a, DATA_TYPE t);
    // занесение идентификатора a в таблицу с типом t

void SemSetType(Tree *Addr, DATA_TYPE t);
     // установить тип t для переменной  по адресу Addr

void SemSetParam(Tree *Addr, int n);
   // установить число формальных параметров n для функции
   // по адресу Addr

void SemControlParam(Tree *Addr, int n);
   // проверить равенство числа формальных параметров n
   // для функции по адресу Addr;

Tree * SemGetType(TypeLex a);
   // найти в таблице переменную с именем a
   // и вернуть ссылку на соответствующий элемент дерева

Tree * SemGetFunct(TypeLex a);
   //  найти в таблице функцию с именем a
   //  и вернуть ссылку на соответствующий элемент дерева

int DupControl(Tree *Addr, TypeLex a);
   // проверка идентификатора a на повторное описание внутри блока
};
#endif
```

Реализацию функций работы с деревом мы уже рассмотрели. Приведем реализацию семантических подпрограмм.

```
// модуль Semant.cpp -- реализация семантических подпрограмм
#include <string.h>
#include "defs.hpp"
#include "Scaner.hpp"
#include "semant.hpp"
 Tree* Tree::Cur=(Tree*)NULL;

void   Tree::SetCur(Tree * a)
// установить  текущий узел дерева
{
Cur=a;
}

Tree * Tree::GetCur(void)
// получить значение текущего узла дерева
{
return Cur;
}

Tree * Tree::SemInclude(TypeLex a, DATA_TYPE t)
// занесение идентификатора a в таблицу с типом t
{
if (DupControl(Cur, a))
       PrintError("Повторное описание идентификатора ",a);
Tree * v;    Node b;
if (t!=TYPE_FUNCT)
   {
   memcpy(b.id,a,strlen(a)+1);   b.DataType=t;    b.Data=NULL;
   b.Param=0;                // количество параметров функции
   Cur->SetLeft (&b);        // сделали вершину - переменную
   Cur = Cur->Left;
   return Cur;
   }
else
   {
   memcpy(b.id,a,strlen(a)+1);   b.DataType=t;    b.Data=NULL;
   b.Param=0;                // количество параметров функции
   Cur->SetLeft (&b);        // сделали вершину - функцию
   Cur = Cur->Left;
   v=Cur;               // это точка возврата после выхода из функции
   memcpy(&b.id,&"",2);   b.DataType=EMPTY;    b.Data=NULL;
   b.Param=0;
   Cur->SetRight (&b);      // сделали пустую вершину
   Cur = Cur->Right;
   return v;
```

```
   }
}

void Tree::SemSetType(Tree* Addr, DATA_TYPE t)
// установить тип t для переменной  по адресу Addr
{
Addr->n->DataType=t;
}

void Tree::SemSetParam(Tree* Addr, int num)
// установить число формальных параметров n для функции по адресу Addr
{
Addr->n->Param=num;
}

void Tree::SemControlParam(Tree *Addr, int num)
// проверить равенство числа формальных параметров значению
// n для функции по адресу Addr
{
if (num!=Addr->n->Param)
    PrintError("Неверное число параметров у функции ",Addr->n->id);
}

Tree * Tree::SemGetType(TypeLex a)
// найти в таблице переменную с именем a
// и вернуть ссылку на соответствующий элемент дерева
{
Tree * v=FindUp(Cur, a);
if (v==NULL)
    PrintError("Отсутствует описание идентификатора ",a);
if (v->n->DataType==TYPE_FUNCT)
    PrintError("Неверное использование вызова функции ",a);
return v;
}

Tree * Tree::SemGetFunct(TypeLex a)
//  найти в таблице функцию с именем a
//  и вернуть ссылку на соответствующий элемент дерева.
{
Tree * v=FindUp(Cur, a);
if (v==NULL)
    PrintError("Отсутствует описание функции ",a);
if (v->n->DataType!=TYPE_FUNCT)
    PrintError("Не является функцией идентификатор ",a);
return v;
}

int Tree::DupControl(Tree* Addr, TypeLex a)
// Проверка идентификатора а на повторное описание внутри блока.
// Поиск осуществляется вверх от вершины Addr.
```

```
{
if (FindUpOneLevel(Addr, a)==NULL) return 0;
return 1;
}
```

Полную реализацию структуры синтаксического анализатора с семантическим контролем здесь мы приводить не будем, т.к. соответствующие вставки вызовов функций выполняются просто. В качестве иллюстрации приведем реализацию только некоторых модулей. Предварительно следует объявить таблицу:

```
Tree * Root;
```

Рассмотрим сначала реализацию заполнения таблицы компилятора. В синтаксической диаграмме "список переменных" следует выполнять занесение переменных в таблицу без типа или с типом, соответствующим типу выражения. Дополнительно необходимо подсчитывать число переменных в списке, т.к. эта конструкция используется не только в операторе описания данных, но и в качестве списка формальных параметров функции.

```
int Z()
// список:                   Z -> Z,I | I
//                           I -> a | a = V
{
TypeLex l; int t, uk1;
int i=0 ; // число переменных в списке
do {
   t=Scaner(l);
   if (t!=TIdent) PrintError("ожидался идентификатор",l);
   i++;
Tree * v=Root->SemInclude(l, TYPE_UNKNOWN );
    // занесение идентификатора l в таблицу с типом TYPE_UNKNOWN
   uk1=GetUK();   t=Scaner(l);
   if (t==TSave)
     {
     DATA_TYPE dt=V();
     Root->SemSetType(v,dt);     // установили переменной тип
     uk1=GetUK(); t=Scaner(l);
     }
   } while(t==TZpt);
PutUK(uk1);
return i;
}
```

Транслятор заносит в таблицу имя функции вместе с числом параметров. После завершения тела функции необходимо предусмотреть возврат на исходную позицию в дереве.

```
void F()
// функция:                   F -> function a (Z) Q
```

```
{
TypeLex l; int t, uk1;
t=Scaner(l);
if (t!=TFunct) PrintError("ожидался символ function",l);
t=Scaner(l);
if (t!=TIdent) PrintError("ожидался идентификатор",l);
Tree * v=Root->SemInclude(l, TYPE_FUNCT);
                  // занесли имя функции в таблицу
int i=0;          // число формальных параметров
t=Scaner(l);
if (t!=TLS) PrintError("ожидался символ (",l);
uk1=GetUK(); t=Scaner(l);
if (t!=TPS)  { PutUK(uk1); i= Z(); t=Scaner(l); }
if (t!=TPS) PrintError("ожидался символ )",l);
Root->SemSetParam(v, i);
   // установить число формальных параметров i для функции
Q();
Root->SetCur(v) ;    // восстановили исходную позицию в дереве
}
```

При трансляции вызова функции контролируется наличие имени функции в таблице, а также соответствие числа фактических и формальных параметров.

```
void H()
// вызов функции:            H -> a ( X )
//                           X -> X , V | V
{
TypeLex l; int t, uk1;
t=Scaner(l);
if (t!=TIdent) PrintError("ожидался идентификатор",l);
Tree * v= Root->SemGetFunct(l);  // поиск имени функции в таблице
int num=0;                       // число фактических параметров
t=Scaner(l);
if (t!=TLS) PrintError("ожидался символ (",l);
uk1=GetUK(); t=Scaner(l);
if (t==TPS)
   {
    Root->SemControlParam(v, num);   // контроль числа параметров
    return; // нет параметров
   }
do {
   num++ ;                    // новый фактический параметр
   V();
   uk1=GetUK(); t=Scaner(l);
   if (t!=TPS) PutUK(uk1);
   } while (t!=TPS);
Root->SemControlParam(v, num);       // контроль числа параметров
if (t!=TPS) PrintError("ожидался знак )",l);
}
```

Обратите внимание, что в *SemInclude* при формировании правого поддерева у функции сначала создается пустой правый потомок с тем, чтобы все внутренние данные этой функции формировались по левым ссылкам от этого пустого элемента.

Принцип пустого элемента придется использовать еще в одном месте: при реализации семантики составного оператора Q. Дело в том, что в соответствии с принципами блочной структуры программы языков программирования все данные, объявленные внутри блока, не доступны вне этого блока. Это значит, что при сканировании открывающейся фигурной скобки надо выполнить те же действия по созданию внутренних данных, что и при трансляции функции. Но тут имеется существенная разница. Дело в том, что у функции есть имя, которое заносится в таблицу, и, следовательно, нет проблем с созданием правого потомка у соответствующей вершины дерева с именем функции. Все последовательно описанные в программе функции являются левыми соседями со своими собственными правыми потомками.

Иное дело блок, у которого отсутствует имя. Если в программе последовательно расположены два или более блоков, то трансляция первого из них приведет к созданию правого потомка, а для последующих блоков это сделать невозможно, т.к. правая ссылка уже занята. Поэтому придется проверить на $NULL$ правую ссылку, и при ненулевом ее значении сначала создать безымянную пустую вершину слева, а только потом от нее строить правое поддерево. Для простоты можно при трансляции блока всегда создавать пустого левого соседа и уже от него — правого потомка. В результате, как это часто бывает в программировании, выигрыш в простоте алгоритма приведет к потерям в памяти, и мы получим увеличение объема данных за счет лишних пустых вершин.

При отладке программы Вы должны предусмотреть такие тесты, которые проверяют все возможные способы описания одноименных переменных во вложенных конструкциях. Например, Ваша программа должна без ошибок обрабатывать следующий текст:

```
<SCRIPT  language="JavaScript">
   <!--
   var a,b,c;
   a=2+d; // неявное описание переменной d
   function fun1(a, d, s)
     //  a и d совпадают по написанию с переменными верхнего уровня
   {
     { var s,ss; // s совпадает с именем параметра
       ss=9-t; dd=-ss;  // ss, dd - новые идентификаторы
     }
     { var s,ss;
       ss=9-t; // ss, t совпадают с именами параллельного уровня
       var eps=0.08, n=100;
     }
     var ss,gg;
   }
   -->
</SCRIPT>
```

Глава 7

$LL(k)$–ГРАММАТИКИ И $LL(k)$–АНАЛИЗАТОРЫ

Метод синтаксических диаграмм обычно применяют для небольших языков программирования. Рассмотрим методы построения синтаксических анализаторов, применяемые к достаточно большим грамматикам. Как известно, различают две стратегии разбора: восходящую (”снизу вверх”) и нисходящую (”сверху вниз”). Эти термины соответствуют способу построения синтаксических деревьев. При нисходящей стратегии разбора дерево строится от корня (аксиомы) вниз к терминальным вершинам. Главная задача при нисходящем разборе — выбор того правила $A \to \phi_i$ из совокупности правил $A \to \phi_1|\phi_2|\ldots|\phi_k$, которое следует применить на рассматриваемом шаге грамматического разбора.

При восходящем разборе дерево строится от терминальных вершин вверх к корню дерева — аксиоме. Главная задача при восходящем разборе — найти тот момент, когда необходимо выполнить редукцию находящейся в верхушке магазина цепочки ϕ по некоторому правилу из множества подходящих правил $A_1 \to \phi$, $A_2 \to \phi$,..., $A_k \to \phi$.

Как уже отмечалось в главе 1, универсальными методами разбора в любой КС–грамматике являются восходящий и нисходящий разбор с возвратами. В соответствии с теоремой о взаимно–однозначном соответствии КС–грамматик и МП–автоматов для произвольной КС–грамматики существует МП–автомат, который выполняет грамматический разбор в этой грамматике. Более того, таких автоматов два: для восходящей и нисходящей стратегии разбора. К сожалению, в общем случае эти автоматы являются недетерминированными, следовательно, универсальные методы требуют очень больших вычислительных ресурсов из-за полного перебора. Поэтому соответствующие алгоритмы неприменимы в качестве основы при практическом построении программы синтаксического анализа транслятора.

Существуют специальные методы разбора, каждый из которых предназначен для грамматик некоторого узкого класса. Рассмотрим нерекурсивные методы анализа, которые позволяют в процессе последовательного чтения входной цепочки вести грамматический разбор без возврата на предшествующие шаги.

Как для процесса порождения терминальных цепочек из аксиомы, так и для процесса грамматического разбора существуют понятия *левый* и *правый* вывод и разбор соответственно. Вывод в КС–грамматике называется левым, если на каждом шаге вывода применяется правило для самого левого нетерминального символа. При правом выводе правило применяется к самому правому нетерминалу. Аналогично *при восходящем левом грамматическом разборе* выполняется редукция для основы —

самой левой простой фразы текущей сентенциальной формы.

Интересно отметить, что левый восходящий разбор соответствует правому выводу. Действительно, каждая очередная редукция в текущей сентенциальной форме $x = x_1 \phi x_2$ выполняется на ее левом конце, в результате редукции будет получена сентенциальная форма $y = x_1 A x_2$, если было применено правило $A \rightarrow \phi$. Причем никакая редукция в подцепочке x_1 невозможна, а подцепочка x_2 еще не подвергалась редукции и, следовательно, состоит из терминальных символов. Но тогда в процессе правого вывода из $y = x_1 A x_2$ в силу терминальности x_2 правило применяется к нетерминалу A. В частности, самая первая редукция выполняется применением терминального правила в левой части цепочки. Но именно это правило применяется последним при правом выводе этой цепочки.

Например, пусть дана КС–грамматика

$$\begin{aligned} G:\quad & S \rightarrow SaA|BA \\ & A \rightarrow aAb|c \\ & B \rightarrow cB|AA. \end{aligned}$$

Тогда, например, правому выводу

$$S \Rightarrow SaA \Rightarrow SaaAb \Rightarrow Saacb \Rightarrow BAaacb \Rightarrow Bcaacb \Rightarrow AAcaacb \Rightarrow Accaacb \Rightarrow cccaacb$$

в G соответствует левый восходящий разбор (см. рис. 7.1):

$$cccaacb \Leftarrow Accaacb \Leftarrow AAcaacb \Leftarrow Bcaacb \Leftarrow BAaacb \Leftarrow Saacb \Leftarrow SaaAb \Leftarrow SaA \Leftarrow S.$$

Нисходящий грамматический разбор полностью соответствует процессу порождения. Следовательно, можно сделать очень важный вывод: *при нисходящем левом грамматическом разборе*, как и при левом выводе, в построенном частичном дереве разбора правило применяется к самому левому нетерминалу разбора. (Заметим, что при нисходящем разборе дерево разбора более естественно называть деревом вывода.) Например, если в дереве разбора, представленном на рис. 7.1, заменим числа 1, 2,...,8, указывающие номера шагов, на 8, 7,..., 1, то получим дерево левого нисходящего грамматического разбора. Это дерево соответствует левому выводу

$$S \Rightarrow SaA \Rightarrow BAaA \Rightarrow AAAaA \Rightarrow cAAaA \Rightarrow ccAaA \Rightarrow cccaA \Rightarrow cccaaAb \Rightarrow cccaacb.$$

Такое большое внимание левому или правому грамматическому разбору мы уделяем не случайно. Дело в том, что программист пишет программу слева направо, последовательно записывая операторы своей программы. Если в программе указаны некоторые описания данных, то эти данные могут использоваться только в последующих за этим описанием операторах.

В процессе программирования неявно предполагается тот факт, что программа последовательно оператор за оператором выполняется также слева направо. Чтобы учесть правила левостороннего написания программы, транслятор также должен читать программу слева направо, последовательно выполняя распознавание конструкций этой программы. Таким образом, *нисходящий синтаксический анализатор должен выполнять левый разбор, соответствующий левому выводу; восходящий синтаксический анализатор должен выполнять левый разбор, соответствующий правому выводу.*

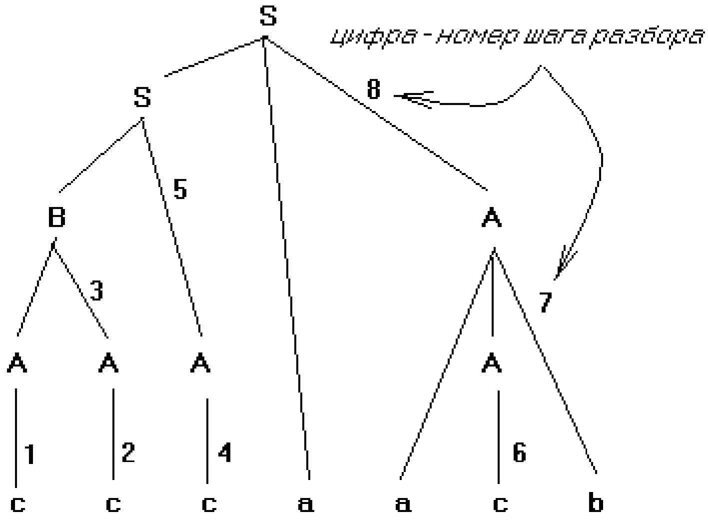


Рис. 7.1: Восходящий синтаксический анализ

7.1 Определение LL(k) - грамматики

Рассмотрим нисходящий левый разбор и выясним условия, при которых разбор можно выполнить за один проход без возвратов, сканируя не более k очередных символов. Очевидно, что эти k очередных символов являются самыми левыми еще не прочитанными и не обработанными символами в исходной цепочке. При нисходящем разборе на каждом шаге решается задача выбора подходящего правила для очередного нетерминала. Хотелось бы определить такие алгоритмы грамматического разбора, когда эти очередные k символов однозначно определяют правило, которое требуется применить на текущем шаге. Например, в КС-грамматике

$$G_1: \quad \begin{array}{l} S \to aSb|A \\ A \to ccA|d \end{array}$$

один очередной отсканированный символ однозначно определяет применяемое правило для каждого нетерминала:

очередной символ	a	b	c	d
нетерминал S	правило $S \to aSb$		правило $S \to A$	правило $S \to A$
нетерминал A			правило $A \to ccA$	правило $A \to d$

В грамматике

$$G_2: \quad \begin{array}{l} S \to aaSb|A|abSa \\ A \to accA|d \end{array}$$

для нетерминала S по одному отсканированному символу уже нельзя определить применяемое при разборе правило. Только два очередных символа позволяют однозначно выбрать правило для S:

очередная пара символов	aa	ab	ac, db, da, $d\natural$
нетерминал S	правило $S \to aaSb$	$S \to abSa$	правило $S \to A$

Приведенные выше таблицы являются примерами *управляющих таблиц* $LL(k)$–*анализатора*. Строка управляющей таблицы соответствует символу в верхушке магазина, а столбец определяется той отсканированной цепочкой длины k, которая помогает в выборе правила КС–грамматики при нисходящем анализе.

Говоря неформально, $LL(k)$–грамматикой называется КС–грамматика, в которой на каждом шаге левого нисходящего разбора k очередных отсканированных символов однозначно определяют правило, применяемое к очередному левому нетерминалу. Таким образом, G_1 — $LL(1)$–грамматика, а G_2 — $LL(2)$–грамматика. Для каждой $LL(k)$–грамматики можно построить детерминированный левый анализатор, выполняющий разбор за время, являющееся линейной функцией длины разбираемой цепочки.

Дадим формальное определение $LL(k)$–грамматики.

Определение 7.1. Грамматика G называется $LL(k)$–грамматикой для некоторого фиксированного k, если для любых двух ее правил $A \to x$ и $A \to y$ из существования двух левых выводов

$$S \overset{*}{\Rightarrow} wAz \Rightarrow wxz \overset{*}{\Rightarrow} wq,$$
$$S \overset{*}{\Rightarrow} wAz \Rightarrow wyz \overset{*}{\Rightarrow} wp,$$

для которых $first_k(q) = first_k(p)$, следует $x = y$.

Говоря менее формально, грамматика G будет $LL(k)$–грамматикой, если для произвольной данной цепочки wAz и первых k символов, выводящихся из Az, существует не более одного правила, которое можно применить к нетерминалу A, чтобы получить правильное дерево разбора.

Очевидно, что максимальная скорость работы транслятора будет при $k = 1$. К сожалению, алгоритмически неразрешимыми являются следующие проблемы, имеющие самое непосредственное отношение к практике построения синтаксических анализаторов:

1) существует ли некоторое k для произвольной КС–грамматики G, такое, что G является $LL(k)$–грамматикой;

2) существует ли для произвольной $LL(k)$–грамматики эквивалентная ей $LL(1)$–грамматика.

Поэтому при построении и преобразовании грамматики языка программирования программист часто должен полагаться на интуицию и знания о некоторых свойствах, нарушающих $LL(k)$–требования. Рассмотрим условия, которые с необходимостью нарушают свойства КС–грамматики быть $LL(1)$–грамматикой.

а) Пусть КС-грамматика является леворекурсивной, т.е. для некоторого нетерминала A имеются правила $A \to Az$ и $A \to y$, следовательно существует вывод

$$A \Rightarrow Az \Rightarrow Azz \Rightarrow Az^* \Rightarrow yz^*.$$

Тогда при непустой цепочке y имеем равенство

$$first_1(Az) = first_1(y)$$

и грамматика не является $LL(1)$–грамматикой. При пустой цепочке y выбор между правилами $A \to Az$ и $A \to y$ должен быть организован по $first_1(Az)$ и $follow_1(A)$, но пересечение этих множеств не пусто.

Следовательно, *необходимо избавиться от левой рекурсии*, заменив правила

$$A \to Az_1|Az_2|...|Az_n|y_1|y_2|...|y_m$$

на совокупность правил

$$A \rightarrow y_1D|y_2D|...|y_mD$$
$$D \rightarrow z_1D|z_2D|...|z_nD|\varepsilon.$$

б) Пусть для какого-либо нетерминала правила имеют одинаковые левые множители,

$$A \rightarrow yx_1|yx_2|...|yx_m|z_1|z_2|...|z_k, y \neq \varepsilon.$$

Построим функции $first_1$ для этих правил и получим, что $first_1(y_i) \cap first_1(y_j) \neq \emptyset$. Непустое пересечение этих множеств означает невозможность выбора применимого правила по очередному отсканированному символу. *Общие множители в правилах необходимо вынести.* Простейший алгоритм устранения общих левых множителей основан на замене правил

$$A \rightarrow yx_1|yx_2|...|yx_m|z_1|z_2|...|z_k$$

на правила

$$A \rightarrow yD|z_1|z_2|...|z_k$$
$$D \rightarrow x_1|x_2|...|x_m.$$

в) Кроме явных общих левых множителей в правилах КС–грамматики могут присутствовать скрытые левые множители, которые проявляют себя в процессе последовательного применения целой совокупности правил. *Скрытые левые множители также следует вынести.* Выявить скрытые левые множители сложнее, чем обнаружить нарушения типа (а) и (б). Однако при некотором опыте анализа КС–грамматик это легко сделать. Например, правила

$$A \rightarrow Bx|y$$
$$B \rightarrow yz$$

можно заменить на правила

$$A \rightarrow yT$$
$$T \rightarrow z|\varepsilon$$
$$B \rightarrow yz.$$

Одинаковые начала правил для разных нетерминальных символов ($A \rightarrow yT$ и $B \rightarrow yz$) при нисходящем разборе не приводят к дополнительным трудностям, т.к. нисходящий разбор основан на определении применяемого правила для конкретного нетерминала, находящегося в верхушке магазина.

7.2 Управляющая таблица для LL(k) - грамматики

В основе алгоритма разбора методом $LL(k)$–грамматик лежит магазин, в который будем записывать правые части применяемых правил. Если в верхушке магазина находится нетерминал, то для $LL(k)$–грамматики анализ верхушки магазина и очередных k отсканированных символов однозначно определяют применяемое правило. Правую часть этого правила необходимо записать в магазин для того, чтобы в дальнейшем проконтролировать наличие всех ожидаемых символов. В соответствии с принципом работы магазина правило записывается в зеркальном отображении. А именно, *зеркальное отображение правой части правила записывается в магазин* по той причине, что магазин работает по принципу $LIFO$, т.е. " последний записанный — первый считанный" (last in — first out). В результате такой записи самый первый символ правой части правила окажется верхним в магазине.

Если в верхушке магазина находится терминал, он сравнивается по типу лексемы с очередным отсканированным символом; при совпадении терминал из верхушки магазина удаляется. Если терминальные символы не совпадают, то регистрируется ошибка "неверный символ lex" или "ожидался символ из верхушки магазина".

Для того, чтобы при достижении конца исходной цепочки анализатор работал по тому же алгоритму, что и в любой другой точке, будем дополнять исходный модуль справа k знаками–ограничителями — символами конца $\natural$. Фактически никакого дополнения не происходит. Если сканер построен правильно, то при достижении конца исходного модуля он остается на последней позиции и при каждом обращении к нему выдает один и тот же тип лексической единицы — "конец".

Перед программированием $LL(k)$–анализатора строится управляющая таблица. Для терминалов в верхушке магазина управляющая таблица представляет собой единичную матрицу. Каждая "1" на главной диагонали означает действие "стереть верхушку магазина и отсканировать новую лексему". Все "0" означают выдачу сообщения об ошибке. В силу тривиального характера этой части управляющей таблицы для терминальных элементов она никогда явно не строится.

Рассмотрим управляющую таблицу для нетерминалов. Пусть строки соответствуют нетерминалам, а столбцы — терминальным цепочкам длины k. В каждой клетке таблицы указывается зеркальное отображение правой части правила и, возможно, некоторая семантическая информация, необходимая для функционирования семантических подпрограмм.

Алгоритм формирования управляющей таблицы основан на определении 7.1 (см. стр. 207): для каждого правила в некотором выводе $S \overset{*}{\Rightarrow} wAz \Rightarrow wxz \overset{*}{\Rightarrow} wq$ необходимо найти $first_k(xz)$. Для построения таблицы последовательно анализируются правила грамматики. Пусть правило имеет вид

$$A \to z,\ first_k(z) = \{x_1, x_2, .., x_n\}.$$

Будем рассматривать цепочки $x_i \in first_k(z)$. Для однозначного определения применяемого правила синтаксический анализатор должен всегда иметь цепочку длины k, поэтому в зависимости от длины цепочек x_i нужно выполнять различные действия по построению таблицы $LL(k)$–анализатора. Имеется всего два варианта: $|x_i| = k$ и $|x_i| < k$.

а) Если длина цепочки x_i равна k, то заполняется клетка таблицы, соответствующая символу A и цепочке x_i:

$$T[A][x_i] = \tilde{z},$$

где $\tilde{z}$ — зеркальное отображение цепочки z.

б) Если длина цепочки x_i меньше k, то вычисляется

$$first_k(x_i \cdot follow_k(A)) = \{y_1, y_2, ..., y_m\}$$

и для любого y_j заполняется клетка таблицы

$$T[A][y_i] = \tilde{z}.$$

Рассмотрим *частный случай этого алгоритма для* $LL(1)$*–грамматики.* Сначала для каждого правила

$$A \to z$$

вычислим $first_1(z) = \{a_1, a_2, .., a_m\}$, где a_1, a_2,.., a_m — терминальные символы и, возможно, пустая цепочка ε. В соответствии с условием (а) заполняем клетки таблицы:

$$T[A][a_i] = \tilde{z},\ a_i \in first_1(z),\ a_i \neq \varepsilon.$$

Если $\varepsilon \in first_1(z)$, то в соответствии с правилом (б) рассматриваем множество

$$follow_1(A) = \{h_1, h_2, ..., h_l\}.$$

Благодаря тому факту, что исходный модуль дополнен справа маркером конца, все $h_i \neq \varepsilon$, поэтому можно заполнить клетки управляющей таблицы

$$T[A][h_i] = \varepsilon,\ h_i \in follow_1(A).$$

Пустые клетки построенной таблицы соответствуют ошибкам, а конфликты правил в некоторой клетке означают, что данная грамматика не является $LL(k)$–грамматикой. Конфликт — это наличие в одной клетке таблицы более одного правила. Правила конфликтуют, если по первым k символам нельзя однозначно определить, какое из данных правил применимо в анализируемом контексте. В этом случае программист должен выбрать один из путей решения возникшей проблемы:

а) считать неприемлемым метод;

б) увеличить k; при этом сразу следует учесть возрастание размеров управляющей таблицы, и как следствие — существенное увеличение объема программы синтаксического анализатора и времени его работы;

в) преобразовать грамматику; здесь следует отметить, что опытный программист сразу построит грамматику так, чтобы не требовались ее явные преобразования;

г) так наложить ограничения на анализируемый язык, чтобы конфликт был ликвидирован;

д) оставить без изменения грамматику и конфликтующие правила, а разрешение конфликтов перенести в программу анализатора, реализуя специальными проверками расширение контекста.

Как правило, совместное применение вариантов (г) и (д) разрешения конфликтов приводят к успешному решению проблемы. Конечно, это возможно только при наличии небольшого числа конфликтов.

Пример 7.1.

$$\begin{array}{ll} G_1: & S \to if(V)\ O | if(V)\ O\ else\ O \\ & O \to S | a = V; \\ & V \to V + A | V - A | A \\ & A \to A * B | A / B | B \\ & B \to a | c | (V). \end{array}$$

Удалим левые множители и левую рекурсию, получим грамматику

$$\begin{array}{ll} G_2: & S \to if(V)\ O\ M \\ & M \to \varepsilon | else\ O \\ & O \to S | a = V; \\ & V \to AR \\ & R \to +AR | -AR | \varepsilon \\ & A \to BF \\ & F \to +BF | /BF | \varepsilon \\ & B \to a | c | (V). \end{array}$$

Вычислим функцию $first_1$ и, поскольку грамматика укорачивающая, функцию $follow_1$:

	S	O	V	A	B	M	R	F
first	if	a if	a c $($	a c $($	a c $($	ε $else$	$+$ $-$ ε	$*$ $/$ ε
follow	$\natural$ $else$	$\natural$ $else$	$;$ $)$	$+$ $-$ $;$ $)$	$*$ $/$ $;$ $)$	$\natural$ else	$;$ $)$	$;$ $)$

Строим управляющую таблицу, столбцы которой соответствуют терминалам, а строки — нетерминальным символам:

	if	$else$	$+$	$-$	$*$	$/$	$($	$)$	a	c	$;$	$\natural$
S	$MO)V(if$											
O	S								$;V=a$			
V							RA		RA	RA		
A							FB		FB	FB		
B							$)V($		a	c		
M		$Oelse$ ε										ε
R			$RA+$	$RA-$				ε			ε	
F					$FB*$	$FB/$		ε			ε	

Таблица строится путем последовательного анализа всех правил грамматики. Для каждого правила $A \to \phi$ строится функция $first_1(\phi)$, анализируется наличие ε в построенном множестве, заполняются соответствующие клетки строки A. Рассмотрим некоторые примеры заполнения клеток таблицы.

1) Правило $S \to if(V)OM$.

$first_1(if(V)OM) = \{if\}$, следовательно, заполняется клетка $T[S][if]$ зеркальным отображением правой части правила: $T[S][if] = MO)V(if$.

2) Правило $M \to \varepsilon$.

$first_1(\varepsilon) = \{\varepsilon\}$, следовательно, используется множество $follow_1(M) = \{\natural, else\}$ и заполняются клетки $T[M][\natural]$ и $T[M][else]$, в которые заносится ε.

3) Правило $M \to elseO$.

$first_1(elseO) = \{else\}$ и заполняется клетка таблицы $T[M][else] = Oelse$.

4) Правило $O \to S$.

$first_1(S) = \{if\}$ и заполняется $T[O][if] = S$.

5) Правило $O \to a = V;$.

$first_1(a = V;) = \{a\}$ и заполняется $T[O][a] =; V = a$.

6) Правило $V \to AR$.

$first(AR) = \{a, c, (\}$ и заполняются три клетки таблицы одним и тем же значением $T[V][a] = T[V][c] = T[V][(] = RA$.

Следует отметить наличие конфликта правил $M \to \varepsilon$ и $M \to elseO$ для нетерминала M: в одной клетке таблицы оказались два правила. В данном случае конфликт устраняется искусственно внесением известного каждому программисту правила использования if и $else$: "каждый $else$ соответствует последнему if".

7.3 LL(k)-анализатор

Использование управляющей таблицы позволяет реализовать алгоритм $LL(k)$–анализатора, представленный на рис. 7.2. В структурной схеме алгоритма через Mag и Lex соответственно обозначены верхушка магазина и отсканированная лексема. Сканер вызывается в следующих двух случаях:

а) в начальный момент для подготовки первого сравнения;

б) в тот момент, когда произошло сравнение терминала, находящегося в верхушке Mag, и отсканированной лексемы.

При несовпадении ожидаемого символа и отсканированной лексемы выдается сообщение об ошибке. Ошибка может быть выдана в соответствии с ситуацией в верхушке Mag и с отсканированной лексемой:

Mag	Lex	Тип ошибки
a	b	Неверный символ b Ожидался a
a	$\natural$	Неожиданный конец программы Ожидался a
$\natural$	b	Лишний текст за логическим концом программы
A	b	Неверный символ b Неверная конструкция A

При реализации семантического уровня языка в отличие от метода диаграмм необходимо связать семантические подпрограммы не с диаграммами, а с правилами и терминалами. Тогда каждому правилу может соответствовать некоторое семантическое условие. Семантическое условие может соответствовать и терминальному символу. Такое соответствие устанавливается с помощью синтаксически–ориентированного перевода, который будет рассмотрен во второй части данного учебного пособия. Сейчас отметим только, что поскольку с каждым правилом грамматики может быть связана некоторая семантическая подпрограмма, указатель на нее должен храниться в управляющей таблице наряду с правой частью правила.

Рассмотренная схема соответствует $LL(k)$–анализатору для $k = 1$. При $k > 1$ необходимо модифицировать алгоритм:

1) перед циклом вместо однократного вызова сканера отсканировать k символов;

2) элемент таблицы определяется верхушкой Mag и отсканированной цепочкой длины k;

3) после успешного сравнения двух терминалов сканируется новый символ, который приписывается в конец текущей цепочки длины k.

7.4 Программа LL(k)-анализатора

В том случае, когда некоторый алгоритм построен по таблице, графу или любому другому формальному определению действий в зависимости от возникшей ситуации, возможны два способа программирования:

1) неявный способ, который основан на встраивании таблицы в программу в виде соответствующих операторов (например, мы уже рассмотрели неявный метод программирования сканера);

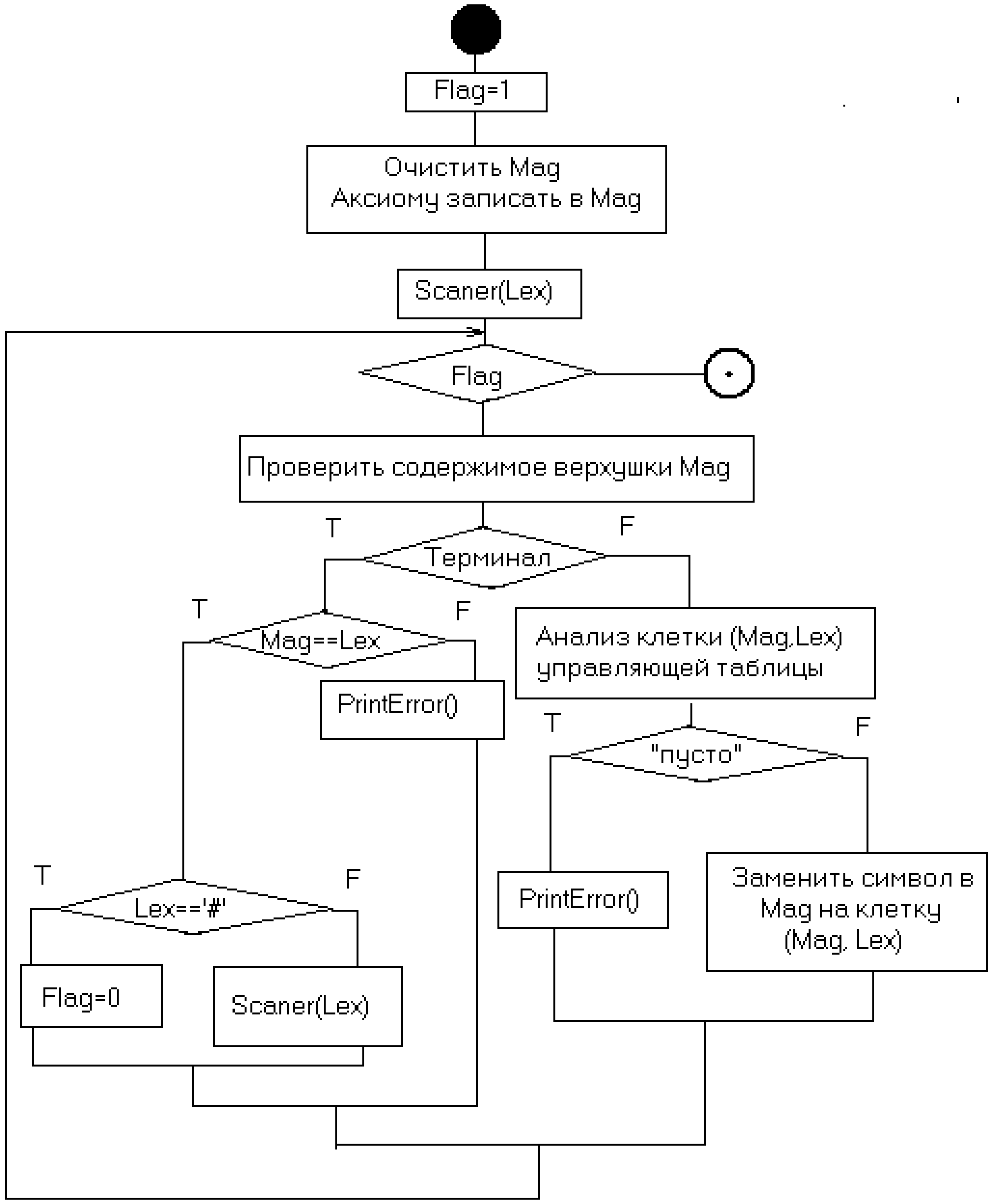


Рис. 7.2: Структурная схема $LL(1)$–анализатора

2) явный способ — реализация универсальной программы, которая имеет в качестве исходной информации таблицу в явной форме, осуществляет в ней поиск соответствующего элемента, а затем выполняет действия, определяемые в таблице.

Достоинства первого способа — высокая скорость, а недостаток — необходимость изменения программы при изменении таблицы. Наоборот, очевидные достоинства второго способа — простая перенастраиваемость программы на изменения в грамматике, а недостаток — большая используемая память под таблицы и низкая скорость работы. Наверное, лучший вариант — использование генератора программ, работающих по первому методу. Правда, в этом случае предварительно необходимо запрограммировать такой генератор.

Любой способ программирования использует магазин, в котором хранятся терминальные и нетерминальные символы, причем нетерминалы можно представить в виде типов с большими целыми значениями этих типов:

```
#define MaxLenMag    1000
                      // максимальный размер магазина
struct TOneSymb
   {              // один символ магазина или клетки таблицы
   unsigned char  Term;          // признак терминала или нетерминала
   unsigned char  Typ;           // тип символа
   TypeLex Lex;                  // изображение лексемы,
    //  (обычно заполняется довольно редко и только для магазина!! )
   } ;

TOneSymb Mag[MaxLenMag]; // магазин - массив  TOneSymb
int i;                   // указатель верхушки магазина
```

Универсальный метод программирования $LL(k)$–анализатора использует таблицу, один элемент которой содержит правую часть правила в зеркальном отображении:

```
#define MaxLenRule  15
             // максимальная длина правой части правила
#define MaxNeterm       100
             // максимальное число нетерминалов
#define MaxTerm         100
             // максимальное число терминалов
struct TCell
   {            // одна клетка таблицы
   unsigned char  l;             // длина правой части правила
   TOneSymb Rule[MaxLenRule];    // правая часть правила
   };

// управляющая таблица:
TCell *Tabl =  new TCell[MaxNeterm*MaxTerm];
     (TCell *) malloc (sizeof(TCell)*MaxNeterm*MaxTerm);
```

Память для организации магазина не зависит от выбора явного или неявного способа программирования анализатора. В магазине нужно хранить только те данные, без которых нельзя в принципе реализовать анализатор. В частности, хранение изображения нетерминала приводит к неоправданному расходу памяти, а используется только на семантическом уровне и только для операндов выражений (идентификаторов и констант). С учетом того факта, что все операнды заносятся в таблицы компилятора, существенно сократить объем памяти под магазин можно, если в магазине хранить не сами операнды, а ссылки на семантические таблицы компилятора. Тогда один элемент магазина будет занимать всего

$$sizeof(Term) + sizeof(Typ) + sizeof(< \text{указатель на таблицу} >).$$

При встраивании таблицы в программу в виде фрагментов этой программы необходимо запрограммировать работу с таблицей в виде операторов записи символов в магазин и чтения символов из магазина. Соответствующий участок программы представляет собой оператор *switch* по элементам в верхушке магазина:

```
switch ( Mag [i].Typ )
{
case  <тип A> : <обработка нетерминала A в соответствии с таблицей>
case  <тип B> : <обработка нетерминала B>
...
}
```

Фрагмент программы, реализующий обработку строки таблицы, представляет собой анализ только заполненных элементов строки таблицы и реализацию записи в магазин требуемой информации. Указанный алгоритм работает до первой ошибки. Для нейтрализации ошибок используются специальные методы, которые будут изложены позже.

Пример 7.2. Рассмотрим строку R из примера 7.1:

	if	$else$	$+$	$-$	$*$	$/$	$($	$)$	a	c	$;$	$\natural$
R			$RA+$	$RA-$				ε			ε	

Фрагмент программы для R имеет вид:

```
case <тип R>:
    if ( t ==< тип + > ) Proc1();
    else
    if ( t ==< тип - > ) Proc2();
    else
    if ( ( t==< тип ) > ) || ( t == < тип ; >) ) Epsilon();
    else       PrintError(...);         //неверный символ Lex
    break;
```

Заполнение магазина удобно организовать специальными функциями, каждая из которых соответствует непустой клетке таблицы, например, функция $Proc1()$ построена для правила $R \to +AR$ и имеет вид:

```
void Proc1(void)
// правило R -> +AR
   {
   Mag[i].Typ=<тип R>;      // заменили нетерминал на R;
             // поскольку R уже в магазине, то этот оператор лишний !!!
   i++; Mag[i].Term=0; Mag[i].Typ=<тип A>;
                           // записали в магазин нетерминал A
   i++; Mag[i].Term=1; Mag[i].Typ=<тип +>;
                           // записали в магазин терминал "+"
   }
```

Функция $Epsilon()$ стирает верхушку магазина и не обязательно должна быть реализована в виде отдельной функции:

```
void Epsilon(void)
// стереть верхушку магазина
{ i--; }
```

7.5 S–анализаторы

Существует подкласс класса $LL(1)$–грамматик, анализ в которых допускает некоторые упрощения. Пусть каждое правило грамматики начинается терминалом, тогда каждый шаг выбора правила предельно прост: если a — текущий входной символ, A — верхний символ магазина и в грамматике имеется правило $A \to ax$, то в магазине надо заменить A на x и отсканировать новый символ.

Допустим теперь, что наряду с правилами вида $A \to ax$ имеются правила с пустой правой частью $A \to \varepsilon$. Поскольку грамматика является $LL(1)$–грамматикой, то один очередной символ должен определить применяемое правило. Тогда при сканировании символа a нужно применить правило $A \to \varepsilon$, если отсутствует правило $A \to ax$.

Для того, чтобы описанный алгоритм однозначно определял применяемой правило, достаточно выполнение следующих требований:

1) каждое правило грамматики либо с пустой правой частью $A \to \varepsilon$, либо начинается терминалом $A \to ax$;

2) правые части правил $A \to ax$ и $A \to by$ для одного нетерминала начинаются разными символами.

Назовем грамматику, удовлетворяющую вышеизложенным требованиям, S–грамматикой. Управляющая таблица для S–грамматик строится в соответствии с двумя правилами:

1) клетка для нетерминала A в верхушке магазина и отсканированного терминала a содержит зеркальное отображение цепочки x тогда и только тогда, когда имеется правило $A \to ax$;

2) клетка для нетерминала A и отсканированного терминала b содержит специальный признак del тогда и только тогда, когда в грамматике имеется правило $A \to \varepsilon$ и отсутствует правило вида $A \to by$. Признак del означает действие "стереть верхушку магазина и не сканировать новый символ".

Алгоритм S–анализатора практически совпадает с алгоритмом $LL(1)$–анализатора, однако скорость работы S–анализатора выше в силу сокращенной формы таблицы.

Пример 7.3. S–грамматика команды *path* MS DOS имеет вид:

$$
\begin{aligned}
G: \quad & S \to pathA \\
& A \to a : B; A | \varepsilon \\
& B \to \ aB | \varepsilon,
\end{aligned}
$$

где a — идентификатор. Тогда управляющая таблица содержит следующие элементы:

	$path$	a	$:$	$;$	$\backslash$	$\natural$
S	A					
A	del	$A; B:$	del	del	del	del
B	del	del	del	del	Ba	del

Разбор цепочки $path\ c :; d : \backslash dir1 \backslash file1;$ выполняется по шагам:

вход	$path$	c	$:$	$;$		d	$:$	$\backslash$	$dir1$	$\backslash$	$file1$	$;$		$\natural$
м														
а														
г														
а			$:$				$:$		a		a			
з			B	B			B	B	B	B	B	B		
и			$;$	$;$	$;$		$;$	$;$	$;$	$:$	$;$	$;$	$;$	
н	S	A	A	A	A	A	A	A	A	A	A	A	A	A
	(1)	(2)	==	(4)	==	(2)	==	(4)	==	(4)	==	(5)	==	(3)

В последней строке таблицы указаны действия, выполняемые S–анализатором: номер применяемого правила или совпадение терминальной верхушки магазина с очередным правилом грамматики.

Этот разбор соответствует выводу

$$
\begin{aligned}
S \Rightarrow \quad & path\ A \Rightarrow path\ a : B; A \\
& \Rightarrow path\ a : \backslash aB; A \\
& \Rightarrow path\ a : \backslash a; A \\
& \Rightarrow path\ a : \backslash a; a : B; A \\
& \Rightarrow path\ a : \backslash a; a : \backslash aB; A \\
& \Rightarrow path\ a : \backslash a; a : \backslash a \backslash aB; A \\
& \Rightarrow path\ a : \backslash a; a : \backslash a \backslash a; A \\
& \Rightarrow path\ a : \backslash a; a : \backslash a \backslash a; .
\end{aligned}
$$

Соответствующая программа разбора имеет вид:

```
#include <stdio.h>
#include <string.h>
#include "defs.hpp"
#include "Scaner.hpp"
#define _S 5000                    // нетерминал S
#define _A 5001                    // нетерминал A
#define _B 5002                    // нетерминал B
```

```
int Mag[1000], i;                  // магазин и его указатель
int type; TypeLex lex;

int main(int argc, char * argv[])
{
if (argc<=1) GetData("input.txt"); // без параметра - файл по умолчанию
else GetData(argv[1]);     // ввести данные
int Flag=1;                        // флаг продолжения разбора
i=0;   Mag[i]=_S;                  // в магазине аксиома
type=Scaner(lex);
while(Flag)
     {
     switch (Mag[i])
{
case _S:
 if (type==TPath) {Mag[i]=_A; type=Scaner(lex); }
    else PrintError("Ожидалось ключевое слово path",lex);
 break;
case _A:
 if (type==TIdent)
   {
   i++; // Mag[i++]= _A;  - лишнее действие записи в Mag
   Mag[i++]= TTZpt;
   Mag[i++]= _B;
   Mag[i]  = TDvoet;
   type=Scaner(lex);
   }
   else i--;      // команда del
   break;
case _B:
 if (type==TSlash)
   {
   i++; //  Mag[i++]= _B; - лишнее действие записи в Mag
   Mag[i]  = TIdent;
   type=Scaner(lex);
   }
   else i--;      // команда del
 break;
case TIdent:
case TSlash:
case TTZpt:
case TDvoet:
 if (Mag[i]==type) {i--; type=Scaner(lex);}
    else PrintError("Неверный символ",lex);
 break;
}
     if ((i<0) && type==TEnd) Flag=0;
              // если магазин пуст и текст закончился, то конец работы
     }
}
```

7.6 Контрольные вопросы к разделу 5

1. Какую стратегию разбора реализует $LL(k)$–анализатор?

2. Какая цель преследовалась при определении $LL(k)$–грамматик?

3. Какие преобразования КС–грамматики необходимо выполнить перед построением управляющей таблицы $LL(k)$–анализатора?

4. Почему $LL(k)$–грамматика не может быть леворекурсивной?

5. Нужно ли избавляться от правой рекурсии в правилах КС–грамматики перед построением управляющей $LL(k)$–таблицы?

6. Допускаются ли для $LL(k)$–грамматики правила с пустой правой частью?

7. Допускаются ли для $LL(k)$–грамматики правила с одинаковыми правыми частями?

8. Почему правила в клетках управляющей таблицы записываются в зеркальном отображении?

9. Какие действия выполняет $LL(k)$–анализатор, если в верхушке магазина находится терминальный символ? Различаются ли эти действия для $LL(1)$ и $LL(2)$–анализаторов?

10. Какие действия выполняет $LL(1)$–анализатор, если в верхушке магазина находится нетерминальный символ?

11. Всегда ли требуется вычисление фунциии $follow$ при заполнении управляющей таблицы $LL(k)$–анализатора? Почему?

12. Объясните причину возникновения конфликтов в $LL(k)$–таблице.

13. Какие действия должен предпринимать программист, если в управляющей таблице $LL(k)$–анализатора появились конфликты?

14. Всегда ли требуется явное устранение конфликтов в $LL(k)$–таблице?

15. Дайте формальное определение $LL(k)$–грамматики.

16. Левое или правое дерево грамматического разбора строит $LL(k)$–анализатор?

17. Какие действия в программе $LL(1)$–анализатора соответствуют пустым клеткам управляющей таблицы?

18. Как заполняется управляющая таблица $LL(k)$–анализатора?

19. Чем явный способ программирования $LL(k)$–анализатора отличается от неявного способа его программирования? Какой анализатор работает быстрее?

20. Чем программа $LL(1)$–анализатора отличается от программы $LL(2)$–анализатора?

7.7 Тесты для самоконтроля к разделу 5

1. Какой разбор строит нисходящий синтаксический анализатор?

Варианты ответов:

а) нисходящий анализатор должен выполнять правый разбор, соответствующий левому выводу.

б) нисходящий анализатор должен выполнять левый разбор, соответствующий левому выводу.

в) нисходящий анализатор должен выполнять левый разбор, соответствующий правому выводу.

г) нисходящий анализатор должен выполнять правый разбор, соответствующий правому выводу.

д) нисходящий анализатор выполняет левый или правый разбор в зависимости от алгоритма разбора и типа грамматики.

Правильный ответ: б.

2. Какой разбор строит восходящий синтаксический анализатор?

Варианты ответов:

а) восходящий анализатор должен выполнять левый разбор, соответствующий правому выводу.

б) восходящий анализатор должен выполнять левый разбор, соответствующий левому выводу.

в) восходящий анализатор должен выполнять правый разбор, соответствующий правому выводу.

г) восходящий анализатор должен выполнять правый разбор, соответствующий левому выводу.

д) восходящий анализатор выполняет левый или правый разбор в зависимости от алгоритма разбора и типа грамматики.

Правильный ответ: а.

3. Для нетерминала A имеются следующие правила:

$$A \to aAcD|bC$$

Какие клетки управляющей таблицы S–анализатора будут заполнены?

Варианты ответов:

а) $T[A][a] = DcAa$; $T[A][b] = Cb$;

б) $T[A][a] = aAcD$; $T[A][b] = bC$;

в) $T[A][a] = DcA$; $T[A][b] = C$;

г) $T[A][a] = AcD$; $T[A][b] = C$:

д) все перечисленные выше варианты неверны, т.к. если КС–грамматика имеет указанные правила, то S–анализатор построить невозможно.

Правильный ответ: в.

4. Для нетерминала A имеются следующие правила:

$$A \to AcD|bC$$

Какие клетки управляющей таблицы $LL(1)$–анализатора будут заполнены?

Варианты ответов:

а) $T[A][a] = DcA$; $T[A][b] = Cb$;

б) $T[A][a] = AcD$; $T[A][b] = bC$;

в) $T[A][a] = DcA$; $T[A][b] = C$;

г) $T[A][a] = AcD$; $T[A][b] = C$:

д) все перечисленные выше варианты неверны, т.к. если КС–грамматика имеет указанные правила, то $LL(1)$–анализатор построить невозможно.

Правильный ответ: д.

5. Дайте определение $LL(1)$–анализатора.

Варианты ответов:

а) Грамматика G называется $LL(k)$–грамматикой для некоторого фиксированного k, если для любых двух ее правил $A \to x$ и $B \to x$ не существуют два левых вывода

$$S \overset{*}{\Rightarrow} wAz \Rightarrow wxz \overset{*}{\Rightarrow} wq,$$
$$S \overset{*}{\Rightarrow} wBz \Rightarrow wxz \overset{*}{\Rightarrow} wp,$$

для которых $first_k(q) = first_k(p)$.

б) Грамматика G называется $LL(k)$–грамматикой для некоторого фиксированного k, если для любого ее правила $A \to x$ и для левого вывода

$$S \overset{*}{\Rightarrow} wAz \Rightarrow wxz \overset{*}{\Rightarrow} wq$$

выполняется условие $first_k(q) = first_k(xz)$.

в) Грамматика G называется $LL(k)$–грамматикой для некоторого фиксированного k, если для любых двух ее правил $A \to x$ и $B \to y$ из существования двух левых выводов

$$S \overset{*}{\Rightarrow} wAz \Rightarrow wxz \overset{*}{\Rightarrow} wq,$$
$$S \overset{*}{\Rightarrow} wBz \Rightarrow wyz \overset{*}{\Rightarrow} wp,$$

для которых $first_k(q) = first_k(p)$, следует $x = y$.

г) Грамматика G называется $LL(k)$–грамматикой для некоторого фиксированного k, если для любых двух ее правил $A \to x$ и $A \to y$ из существования двух левых выводов

$$S \overset{*}{\Rightarrow} wAz \Rightarrow wxz \overset{*}{\Rightarrow} wq,$$
$$S \overset{*}{\Rightarrow} wAz \Rightarrow wyz \overset{*}{\Rightarrow} wp,$$

для которых $first_k(q) = first_k(p)$, следует $x = y$.

Правильный ответ: г.

7.8 Упражнения к разделу

7.8.1 Задание

Цель данного задания – программирование $LL(1)$–анализатора. Чтобы выполнить задание, сначала нужно преобразовать грамматику к форме $LL(k)$-грамматики, при наличии конфликтов определить действия по их разрешению. Затем можно приступать к программированию синтаксического анализатора. Работу над заданием следует организовать последовательно выполняя следующие операции.

1. Рассмотрите КС–грамматику Вашего языка программирования. Проанализируйте правила этой грамматики с целью обнаружения несовместимости с требованиями, предъявляемыми к $LL(1)$–грамматикам.

2. Устраните обнаруженные недостатки, преобразуя КС–грамматику предназначенными для этого методами.

3. Постройте функции $first$ и $follow$ для полученной КС-грамматики.

4. Постройте управляющую таблицу $LL(1)$–анализатора.

5. Проанализируйте управляющую таблицу $LL(1)$–анализатора, которую Вы построили, выделите все конфликты в этой таблице.

6. Для каждого конфликта в управляющей таблице определите методы его устранения:

- преобразование КС–грамматики;
- расширение контекста в программе анализатора;
- внесение ограничений в язык программирования, компилятор с которого Вы разрабатываете.

7. Для каждого конфликта реализуйте выбранный метод его устранения.

8. Результатом Вашей работы должна стать бесконфликтная таблица, готовая для ее программирования:

– либо каждая клетка содержит не более одного правила,

– либо в некоторой клетке имеется несколько правил, но однозначно определены условия выбора правила (при этом допускается анализ ограниченного контекста или ограниченной цепочки в верхушке магазина).

9. Определите тип данных для представления одного элемента магазина синтаксического анализатора. Приведите программное описание магазина.

10. Напишите основной цикл алгоритма $LL(1)$–анализатора.

11. Для каждого нетерминала Вашей КС–грамматики определите участок программы, соответствующий этому нетерминалу.

12. Постройте фрагменты программы для обработки каждого нетерминала в соответствии с управляющей таблицей.

13. Если конфликтов в управляющей таблице нет, то Вы закончили программу анализатора. в противном случае Вам необходимо встроить в программу разрешение конфликтов в соответствии с предложенными Вами методами (см. п.6 данного задания).

14. Теперь Вам нужно построить проект, включающий

- стандартные определения;
- сканер;
- $LL(1)$–анализатор.

15. В заключение Вам необходимо отладить программу на правильных и ошибочных текстах программ.

7.8.2 Пример выполнения задания

В упражнении к разделу 5 мы построили синтаксический уровень КС–грамматики:

$$
\begin{array}{lll}
G_J: & S \to & <\textbf{SCRIPT language} =''\ \textbf{JavaScript}''> \\
 & & <!--T--> \\
 & & </\textbf{SCRIPT}> \\
 & T \to & T\ W|\varepsilon \\
 & W \to & D|F \\
 & D \to & \textbf{var}\ Z; \\
 & Z \to & Z,\ I|I \\
 & I \to & a \mid a{=}V \\
 & F \to & \textbf{function}\ (Z)\ Q \\
 & Q \to & \{K\} \\
 & K \to & KD|KO|\varepsilon \\
 & O \to & P\textbf{;}\ |Q|H\textbf{;}\ |U|M|\ ; \\
 & U \to & \textbf{for}\ (P;V;P)\ O
\end{array}
$$

$$
\begin{array}{ll}
M \to & \textbf{if } (V)\ O | \textbf{if } (V)\ O \textbf{ else } O \\
P \to & N{=}V \\
N \to & N.a|a \\
V \to & V{>}A|V{>=}A|V{<}A|V{<=}A|V{==}A|V!=A|{+}A|{-}A|A \\
A \to & A{+}B|A{-}B|B \\
B \to & B{*}E|B/E|E \\
E \to & N|C|H|(V) \\
H \to & a(X)|a() \\
X \to & X,V|V \\
C \to & c_1|c_2|c_3|c_4
\end{array}
$$

Сразу можно заметить, что эта грамматика не удовлетворяет требованиям к $LL(1)$–грамматике: она леворекурсивна и содержит одинаковые левые множители для некоторых нетерминалов. Поэтому придется устранить замеченные недостатки. Для нетерминалов T, N, K и Z можно просто переставить соответствующие фрагменты. Строго говоря, леворекурсивный или праворекурсивный характер правил определяет структуру соответствующей программы. Поэтому, чтобы сохранить структуру программы, необходимо устранять левую рекурсию по правилам преобразования КС–грамматик. Однако при нисходящем анализе иногда можно просто переставить левый и правый нетерминал в рекурсивном правиле. Это касается только таких конструкций, которые построены с помощью итерации из последовательности одинаковых простых конструкций более низкого уровня. Например, конструкция <*последовательность операторов*> построена из конструкций <*оператор*>:

$$
\begin{array}{l}
<\text{ последовательность операторов }>\to \\
\quad <\text{ последовательность операторов }><\text{ оператор }>|\varepsilon.
\end{array}
$$

Следует переставить нетерминалы:

$$
\begin{array}{l}
<\text{ последовательность операторов }>\to \\
\quad <\text{ оператор }><\text{ последовательность операторов }>|\varepsilon.
\end{array}
$$

В результате такой перестановки *при нисходящем анализе будет последовательно разворачиваться каждый очередной оператор.* При восходящем анализе такая перестановка недопустима, т.к. в этом случае придется сначала редуцировать все отдельно взятые операторы до нетерминала <*оператор*>, а только затем на правом конце всей последовательности выполнить редукцию к нетерминалу <*последовательность операторов*>.

Для нетерминалов, определяющих выражения, такая перестановка невозможна при любой стратегии разбора, т.к. в соответствии с правилами выражение вычисляется слева направо, а, следовательно, в таком же порядке нужно выполнять редукцию и всех подвыражений выражения в целом. Поэтому для нетерминалов V, A, B мы воспользуемся стандартными алгоритмами устранения левой рекурсии.

В результате получим грамматику

$$
\begin{array}{lll}
G_J: & S \to & <\textbf{SCRIPT language} =''\ \textbf{JavaScript}''> \\
 & & <!--T--> \\
 & & </\textbf{SCRIPT}> \\
 & T \to & W\ T|\varepsilon \\
 & W \to & D|F
\end{array}
$$

$$
\begin{array}{ll}
D \to & \textbf{var}\ Z; \\
Z \to & I Z_1 \\
Z_1 \to & ,\ I\ Z_1 | \varepsilon \\
I \to & a\ I_1 \\
I_1 \to & =V|\varepsilon \\
F \to & \textbf{function}\ (Z)\ Q \\
Q \to & \{K\} \\
K \to & DK|OK|\varepsilon \\
O \to & P\textbf{;}\ \ |\ Q\ |\ H\ ;\ |\ U\ |\ M|\ ; \\
U \to & \textbf{for}\ (P;V;P)\ O \\
M \to & \textbf{if}\ (V)\ OM_1 \\
M_1 \to & \varepsilon\ |\ \textbf{else}\ O \\
P \to & N = V \\
N \to & aN_1 \\
N_1 \to & .\ a\ N_1|\varepsilon \\
V \to & AV_1|+AV_1|-AV_1 \\
V_1 \to & >AV_1|>=AV_1|<AV_1|<=AV_1|==AV_1|V!=AV_1|\varepsilon \\
A \to & BA_1 \\
A_1 \to & +BA_1|-BA_1|\varepsilon \\
B \to & EB_1 \\
B_1 \to & *EB_1|/EB_1|\varepsilon \\
E \to & N|C|H|(V) \\
H \to & a(H_1 \\
H_1 \to & X)\ |\ ) \\
X \to & V\ X_1 \\
X_1 \to & ,\ VX_1|\varepsilon \\
C \to & c_1|c_2|c_3|c_4
\end{array}
$$

Построим функции $first$ и $follow$ для полученной грамматики (см. рис. 7.3), а затем построим управляющую таблицу (см. рис. 7.4).

Анализ управляющей таблицы показывает наличие двух конфликтов:

1) для нетерминала E и символа a в верхушке магазина необходимо выбрать одно из двух правил $E \to H$ или $E \to N$;

2) для нетерминала O и символа a в верхушке магазина необходимо выбрать одно из двух правил $O \to H$ или $O \to P$.

Оба эти конфликта имеют одну и ту же причину: идентификатором может начинаться или операнд выражения, или левая часть оператора присваивания, или вызов функции. В данном случае нет смысла заниматься преобразованием КС–грамматики с целью устранения конфликта. Достаточно воспользоваться функцией $first_2()$ и отсканировать еще один символ, чтобы определить требуемую конструкцию: если следующий отсканированный символ является открывающейся скобкой, то это вызов функции, следовательно, в магазин в качестве правой части правила грамматики необходимо записать цепочку H. В противном случае нужно перейти к анализу операнда, т.е. записать в магазин правую часть правила $E \to N$ или $O \to P$ для символов в верхушке магазина E или O соответственно.

После того, как построена таблица и разрешены конфликты, можно приступать к программированию синтаксического анализатора. Как и при выполнения задания к главе 3, будем использовать следующие программные модули:

– $defs.hpp$ – описание типов данных и констант,

– $Scaner.hpp$, $Scaner.cpp$ – сканер,

– *ll*1.*hpp*, *ll*1.*cpp* – программа LL(1)–анализатора,

– *trans.cpp* – главная программа транслятора.

В проект включаются *Scaner.cpp*, *ll*1.*cpp*, *trans.cpp*.

```
//**************************************************************
//  trans.cpp -- Главная программа транслятора,  в которой
//              работает LL(1)-анализатор до первой ошибки
//**************************************************************
#include <stdio.h>
#include <string.h>
#include "defs.hpp"
#include "Scaner.hpp"
#include "LL1.hpp"

int main(int argc, char * argv[])
{
int type; TypeLex l;
if (argc<=1) GetData("input.txt");// без параметра - файл по умолчанию
        else GetData(argv[1]);    // ввести данные

PutUK(0);   // поставили указатель на начало исходного модуля
LL_1();
return 0;
}

//**************************************************************
//  LL1.cpp --  LL(1)-анализатор до первой ошибки
//**************************************************************
#include <stdio.h>
#include <string.h>
#include "defs.hpp"
#include "Scaner.hpp"
#include "LL1.hpp"

int m[5000],z=0;
        // магазин  LL(1)-анализатора и указатель магазина

void epsilon()
//**************************************************************
// обработка правила с пустой правой частью
//**************************************************************
{z--;}

int LL_1(void)   {
//**************************************************************
// функция синтаксического анализатора
//**************************************************************
int t, fl=1, i, uk1, ttt;
TLex l, lll;
char sss[30];
```

```
m[z]=neterm_S;       // все нетерминалы указаны в defs.hpp
t=Scaner(l);
while(fl)
   {
   if(m[z] <= MaxTypeTerminal) //в верхушке магазина терминал?
      {     // в верхущке магазина терминал
      if(m[z] == t)
         {
         // верхушка совпадает с отсканированным терминалом
         if(t == TEnd)  fl=0; // конец работы
         else  {
               t=Scaner(l);  // сканируем новый символ и
               z--;          // стираем верхушку магазина
               }
         }
       else
         {              // обнаружена ошибка
         PrintError("Неверный символ ",l);
         return -1;     // ожидался символ типа m[z], а не l
         }
      }
    else
      {    // в верхущке магазина нетерминал
      switch(m[z])    // по нетерминалу в верхушке магазина
         { // выполним действия в соответствии таблицей
          case neterm_S :
               // S -> <script ...> <!--  T --> </script>
                         m[z++] = TLT;
                         m[z++] = TScript;
                         m[z++] = TLang;
                         m[z++] = TSave;
                         m[z++] = TConsChar;
                         m[z++] = TGT;
                         m[z++] = TCom1;
                         m[z++] = neterm_T;
                         m[z++] = TCom2;
                         m[z++] = TTegEnd;
                         m[z++] = TScript;
                         m[z++] = TGT;
                         break;

          case neterm_T :
               //   T -> W T | eps
                         if (t == TCom2) epsilon();
                         else {
                              m[z++] = neterm_T;
                              m[z++] = neterm_W;
                              }
                         break;
          case neterm_W :
```

```
    //   W -> D | F
              if (t == TFunct) m[z++] = neterm_F;
              else             m[z++] = neterm_D;
              break;
case neterm_D :
    //   D -> var Z ;
              if (t == TVar)
                 {
                 m[z++] = TTZpt;
                 m[z++] = neterm_Z;
                 m[z++] = TVar;
                 }
              else
                 {
                 PrintError("неверный символ",l);
                 return -1;
                 }
              break;
case neterm_Z :
    //   Z -> I Z1
              m[z++] = neterm_Z1;
              m[z++] = neterm_I;
              break;
case neterm_Z1 :
    //   Z1 -> , I Z1 | eps
              if (t == TZpt)
                 {
                 m[z++] = neterm_Z1;
                 m[z++] = neterm_I;
                 m[z++] = TZpt;
                 }
              else epsilon();
              break;
case neterm_I :
    //   I -> a I1
              m[z++] = neterm_I1;
              m[z++] = TIdent;
              break;
case neterm_I1 :
    //   I1 -> =V | eps
              if (t == TSave)
                 {
                 m[z++] = neterm_V;
                 m[z++] = TSave;
                 }
              else epsilon();
              break;
case neterm_F :
    //   F -> function (Z) Q
              m[z++] = neterm_Q;
```

```
               m[z++] = TPS;
               m[z++] = neterm_Z;
               m[z++] = TLS;
               m[z++] = TFunct;
               break;
case neterm_Q :
     //   Q -> {K}
               m[z++] = TFPS;
               m[z++] = neterm_K;
               m[z++] = TFLS;
case neterm_K : switch(t){
     //   K -> DK | OK | eps
                case TVar :   //    K -> DK
                          m[z++] = neterm_K;
                          m[z++] = neterm_D;
                          break;
                case TIdent :
                case TFLS :
                case TFor :
                case TIf :
                case TZpt :  //     K -> OK
                          m[z++] = neterm_K;
                          m[z++] = neterm_O;
                          break;
                default: epsilon();
                }
                break;
case neterm_O :
     switch(t){
     //   O- >P; | Q | H; | U | M | ;
       case TIdent: // неоднозначность
           uk1 = GetUK(); // O->P; или O->H
           ttt = Scaner(lll);
           PutUK(uk1);
           if (ttt == TLS)
              {  // O -> H
              m[z++] = neterm_H;
              }
           else
              {   // O -> P;
              m[z++] = TTzpt;
              m[z++] = neterm_P;
              }
           break;
       case TFLS :
                m[z++] = neterm_Q;
                break;
       case TIf :
                m[z++] = neterm_M;
                break;
```

```
            case TFor :
                    m[z++] = neterm_U;
                    break;
            case TTzpt :
                    m[z++] = TTzpt;
                    break;
            default: PrintError("неверный символ",l);
                   return -1;
            }
            break;
     case neterm_U :
          //   U -> for (P;V;P) O
                 m[z++] = neterm_O;
                 m[z++] = TPS;
                 m[z++] = neterm_P;
                 m[z++] = TTzpt;
                 m[z++] = neterm_V;
                 m[z++] = TTzpt;
                 m[z++] = neterm_P;
                 m[z++] = TLS;
                 m[z++] = TFor;
                 break;
       case neterm_M :
            //   M -> if (V) O M1
                 m[z++] = neterm_M1;
                 m[z++] = neterm_O;
                 m[z++] = TPS;
                 m[z++] = neterm_V;
                 m[z++] = TLS;
                 m[z++] = TIf;
                 break;
     case neterm_M1 :
          //   M1 -> eps | else O
                 if (t != TElse)
                        epsilon();
                   else {
                        m[z++] = neterm_O;
                        m[z++] = TElse;
                        }
                   break;
     case neterm_P :
          //   P -> N = V
                 m[z++] = neterm_V;
                 m[z++] = TSave;
                 m[z++] = neterm_N;
                 break;
     case neterm_N :
          //   N -> a N1
                 m[z++] = neterm_N1;
                 m[z++] = TIdent;
```

```
            break;
case neterm_N1 :
     //   N1 -> .a N1 | eps
            if (t != TToch)
                   epsilon();
            else   {
                   m[z++] = neterm_N1;
                   m[z++] = TIdent;
                   m[z++] = TToch;
                   }
              break;
case neterm_V :
     //   V-> A V1 | + A V1 | - A V1
            if (t == TPlus)
                   {
                   m[z++] = neterm_V1;
                   m[z++] = neterm_A;
                   m[z++] = TPlus;
                   }
            else
            if (t == Tminus)
                   {
                   m[z++] = neterm_V1;
                   m[z++] = neterm_A;
                   m[z++] = TMinus;
                   }
              else {
                   m[z++] = neterm_V1;
                   m[z++] = neterm_A;
                   }
              break;
case neterm_V1 :
     //   V1 -> "сравн" A V1 | eps
            if ((t <= TNEQ)  &&  (t>=TLT) )
                   {
                   m[z++] = neterm_V1;
                   m[z++] = neterm_A;
                   m[z++] = t;
                   }
            else epsilon();
            break;
case neterm_A :
     //   A -> B A1
            m[z++] = neterm_A1;
            m[z++] = neterm_B;
            break;
case neterm_A1 :
     //   A1 -> + B A1 | -B A1 | eps
            if (t == TPlus)
                   {
```

```
                 m[z++] = neterm_A1;
                 m[z++] = neterm_B;
                 m[z++] = TPlus;
                 }
          else
          if (t == TMinus)
                 {
                 m[z++] = neterm_A1;
                 m[z++] = neterm_B;
                 m[z++] = TMinus;
                 }
          else epsilon();
          break;
case neterm_B : switch(t){
    //   B -> E B1
          m[z++] = neterm_B1;
          m[z++] = neterm_E;
          break;
case neterm_B1 :
     //   B1 -> *E B1 | /E B1 | eps
          if (t == TMult)
                 {
                 m[z++]=neterm_A1;
                 m[z++]=neterm_B;
                 m[z++]=TMult;
                 }
          if (t == TDiv)
                 {
                 m[z++]=neterm_A1;
                 m[z++]=neterm_B;
                 m[z++]=TDiv;
                 }
          else epsilon();
          break;
case neterm_E : switch(t){
     //   E -> N | C | H | (V)
         case TIdent :     // неоднозначность
           uk1 = GetUK();    // E->N или E->H
           ttt = Scaner(lll);
           PutUK(uk1);
           if (ttt == TLS)     // E -> H
              m[z++] = neterm_H;
           else              // O -> N
              m[z++] = neterm_N;
           break;
         case TConsChar:
         case TConsInt:
         case TConsFloat:
         case TConsExp:
                  m[z++] = neterm_C;
```

```
                        break;
                case TLS :
                        m[z++] = TPS;
                        m[z++] = neterm_N;
                        m[z++] = TLS;
                        break;
                }
                break;
          case neterm_H :
                //   H -> a( H1
                     m[z++] = neterm_H1;
                     m[z++] = TLS;
                     m[z++] = TIdent;
                     break;
          case neterm_H1 :
               //   H1 -> X) | )
                      if (t == TPS)
                             m[z++]=TPS;
                        else {
                             m[z++]=TPS;
                             m[z++]=neterm_O;
                             }
                        break;
           case neterm_X :
                //   X -> V X1
                        m[z++] = neterm_X1;
                        m[z++] = neterm_V;
                        break;
           case neterm_X1 : switch(t){
                //   X1 -> ,V X1 | eps
                        if (t == TZpt)
                             {
                             m[z++] = neterm_X1;
                             m[z++] = neterm_V;
                             m[z++] = TZpt;
                             }
                             }
                        else epsilon();
                        break;
           case neterm_C :
                //   C -> c1 | c2| c3 | c4
                    if (  (t == TConsChar) || (t == TConsInt)
                        ||(t == TConsFloat) ||(t == TConsExp) )
                             m[z++]=t;
                    break;
          }  // конец switch по верхушке магазина
       }
    z--;
    }        // конец цикла обработки магазина
 return 1;   // нормальный выход - синтаксический анализ закончен
```

```
}
```

Следует сделать некоторые комментарии к тексту программы анализатора.

1) Если для некоторого нетерминала имеется единственное правило и это правило имеет непустую правую часть, то нет необходимости проверять текущий отсканированный символ на совпадение с терминалами в соответствии с управляющей таблицей. Достаточно просто заменить нетерминал в магазине на правую часть правила. Если символ был неправильный и имела место ошибка, то эта ошибка все равно будет обнаружена в тот момент, когда в верхушке магазина окажется терминальный символ. Соответствующий фрагмент программы Вы можете увидеть, например, для нетерминалов S, D, F, Q. Таким образом, программа анализатора становится короче, но выполняет все требуемые действия.

2) Аналогично можно упростить программирование правил с пустой правой частью. Если для некоторого нетерминала A имеется правило $A \to \varepsilon$, то действия анализатора по применению этого правила заключаются в том, что стирается символ A из верхушки магазина. Допустим, что вместо терминала из множества $follow_1(A)$ отсканирован некоторый другой символ. Если мы просто сотрем символ A в верхушке магазина, то ошибка будет обнаружена на следующем шаге. Именно таким способом в приведенной выше программе анализатора обрабатываются нетерминалы I_1, K, N_1, V_1 и др.

3) Каждая строка управляющей таблицы программируется с помощью операторов if или $switch$ в зависимости от числа альтернатив. Например, для нетерминала N_1 всего два варианта и лучше его записать с помощью оператора if. Для нетерминала K выбор правил определяется шестью разными терминалами, поэтому его проще записать с помощью оператора $switch$.

4) Иногда выбор правила проще осуществить по множеству $follow$, если это множество содержит один или два символа. Например, при программировании правил для нетерминала

$$T \to WT|\varepsilon$$

множество $follow_1(T)$ содержит только один символ. Следовательно, при выполнении только одного сравнения можно выбрать правило $T \to \varepsilon$, а правило $T \to WT$ будет выбираться по альтернативной ветви условного оператора.

нетерминал	$first_1$	$follow_1$
A	(, c_1, c_2, c_3, c_4, a	!=,), +, „ -, ;, <, <= ==, >, >=
A_1	+, - ε	!=,), +, „ -, ;, <, <= ==, >, >=
B	(, c_1, c_2, c_3, c_4, a	!=,), *, +, -, /, ;, <, <=, ==, >, >=
B_1	*, /, ε	!=,), *, +, -, /, ;, <, <=, ==, >, >=
C	c_1, c_2, c_3, c_4	!=,), +, „ -, ;, <, <= ==, >, >=
D	var	(, +, -, c_1, c_2, c_3, c_4, ;, <, a, for, function, if, var, {, }
E	(, c_1, c_2, c_3, c_4, a	!,), *, +, -, /, ;, <, <=, ==, >, >=
F	function	(, +, -, c_1, c_2, c_3, c_4, ;, <, a, for, function, if, var, {
H	a	(,), *, +, -, /, c_1, c_2, c_3, c_4, ;, <, <=, ==, >, >=, !=, a, for, else, function, if, var, {, }, }
H_1	(, +, -, c_1, c_2, c_3, c_4, a,)	
I	a	), , , ;
I_1	, , ε	), „ ;
K	ε, ;, a, for, if, var, {	}
M	if	(, +, -, c_1, c_2, c_3, c_4, ;, <, a, for, else, function, if, var, {, }
M_1	ε, else	
N	a	!=,), *, +, -, ., /, ;, <, <=, =, >, >=, ==
N_1	=, ε	!=,), *, +, -, ., /, ;, <, <=, =, >, >=, ==
O	;, a, for, if, {	(, +, -, c_1, c_2, c_3, c_4, ;, <, a, for, else, function, if, var, {, }
P	a	), ;
Q	{	(, +, -, c_1, c_2, c_3, c_4, ;, <, a, for, else, function, if, var, {, }
S	<	$\natural$
T	ε, function, var	-
U	for	(, +, -, c_1, c_2, c_3, c_4, ;, <, a, for, else, function, if, var, {, }
V	(, +, -, c_1, c_2, c_3, c_4, a	!=,), „ ;, <, <=, ==, >, >=
V_1	<, <=, >, >=, !=, ==, ε	!=,), „ ;, <, <=, ==, >, >=
W	function, var	-
X	(, +, -, c_1, c_2, c_3, c_4, a	), ,
X_1	,, ε	), ,
Z	a	), „ ;
Z_1	, , ε	), ,, ;

Рис. 7.3: Таблица значений функции first и функции follow

Нетерминал в магазине	Терминал на входе	Правая часть правила которая должна быть занесена в магазин
S	<SCRIPT	<SCRIPT language = JavaScript > <!– T –> </SCRIPT >
T	var	$W\ T$
T	function	$W\ T$
T	–>	ε
W	var	D
W	function	F
D	var	var Z ;
Z	a	$I\ Z_1$
Z_1	,	$,IZ_1$
Z_1	;	ε
F	function	function (Z) Q
Q	{	{ K }
K	var	$D\ K$
K	;	$O\ K$
K	{	$O\ K$
K	a	$O\ K$
K	for	$O\ K$
K	if	$O\ K$
K	}	ε
O	a	P ;
O	{	Q
O	a	H
O	for	U
O	if	M
O	;	;
U	for	for (P ; V ; P) O
M	if	if (V) $O\ M_1$
M_1	}	ε
M_1	else	ε
M_1	;	ε
M_1	{	ε
M_1	a	ε
M_1	for	ε
M_1	if	ε
M_1	var	ε
M_1	else	else O
P	a	$N = V$
N	a	aN_1
N_1	.	$,aN_1$
N_1	=	ε
N_1	*	ε

Рис. 7.4: Управляющая таблица LL(1) – анализатора (лист 1)

Нетерминал в магазине	Терминал на входе	Правая часть правила которая должна быть занесена в магазин
N_1	/	ε
N_1	-	ε
N_1	$+$	ε
N_1	)	ε
V	(	$A\ V_1$
V	a	$A\ V_1$
V	c_2	$A\ V_1$
V	c_3	$A\ V_1$
V	c_4	$A\ V_1$
V	c_1	$A\ V_1$
V	$+$	$+\ A\ V_1$
V	-	- $A\ V_1$
V_1	$>$	$>\ A\ V_1$
V_1	$>=$	$>=\ A\ V_1$
V_1	$<$	$<\ A\ V_1$
V_1	$<=$	$<=\ A\ V_1$
V_1	$==$	$==\ A\ V_1$
V_1	$+$	V $!=\ A\ V_1$
V_1	-	V $!=\ A\ V_1$
V_1	(	V $!=\ A\ V_1$
V_1	c_2	V $!=\ A\ V_1$
V_1	c_3	V $!=\ A\ V_1$
V_1	c_4	V $!=\ A\ V_1$
V_1	c_1	V $!=\ A\ V_1$
V_1	a	V $!=\ A\ V_1$
V_1	;	ε
V_1	)	ε
V_1	$!=$	ε
V_1	,	ε
A	(	B A_1
A	c_2	B A_1
A	c_3	B A_1
A	c_4	B A_1
A	c_1	B A_1
A	a	B A_1
A_1	$+$	$+$ B A_1
A_1	-	- B A_1
A_1	$>=$	ε
A_1	$<$	ε
A_1	$<=$	ε
A_1	$==$	ε
A_1	;	ε

Рис. 7.5: Управляющая таблица LL(1) – анализатора (лист 2)

Нетерминал в магазине	Терминал на входе	Правая часть правила которая должна быть занесена в магазин
A_1	)	ε
A_1	!=	ε
A_1	,	ε
A_1	$>$	ε
A_1	$+$	ε
A_1	-	ε
A_1	(	ε
A_1	c_2	ε
A_1	c_3	ε
A_1	c_4	ε
A_1	c_1	ε
A_1	a	ε
B	(	E B_1
B	a	E B_1
B	c_2	E B_1
B	c_3	E B_1
B	c_4	E B_1
B	c_1	E B_1
B_1	*	* E B_1
B_1	/	/ E B_1
B_1	-	ε
B_1	$<$	ε
B_1	$<=$	ε
B_1	$==$	ε
B_1	;	ε
B_1	)	ε
B_1	!=	ε
B_1	,	ε
B_1	$+$	ε
B_1	$>$	ε
B_1	$>=$	ε
B_1	(	ε
B_1	c_2	ε
B_1	c_3	ε
B_1	c_4	ε
B_1	c_1	ε
B_1	a	ε
E	a	N
E	c_1	C
E	c_2	C
E	c_3	C
E	c_4	C

Рис. 7.6: Управляющая таблица LL(1) – анализатора (лист 3)

Нетерминал в магазине	Терминал на входе	Правая часть правила которая должна быть занесена в магазин
E	a	H
E	(	(V)
H	a	A_1 (H_1
H_1	+	X)
H_1	-	X)
H_1	(	X)
H_1	c_2	X)
H_1	c_3	X)
H_1	c_4	X)
H_1	c_1	X)
H_1	a	X)
H_1	)	)
X	+	V , X
X	-	V , X
X	(	V , X
X	c_2	V , X
X	c_3	V , X
X	c_4	V , X
X	c_1	V , X
X	a	V , X
X	+	V
X	-	V
X	(	V
X	c_2	V
X	c_3	V
X	c_4	V
X	c_1	V
X	a	V
C	c_1	c_1
C	c_2	c_2
C	c_3	c_3
C	c_4	c_4
I	a	aI_1
I_1	=	$= V$
I_1	,	ε
I_1	;	ε

Рис. 7.7: Управляющая таблица LL(1) – анализатора (лист 4)

Глава 8

АНАЛИЗАТОРЫ ПРЕДШЕСТВОВАНИЯ

8.1 Простое предшествование

8.1.1 Понятие отношений предшествования

Рассмотрим восходящий анализ. В силу известной теоремы о взаимно–однозначном соответствии КС–языков и МП–автоматов, нам придется использовать магазин в качестве рабочей памяти. Как и всегда, текст транслируемой программы сканируется слева направо, следовательно, при восходящей стратегии разбора главная задача — найти в магазине основу и редуцировать ее в соответствии с правилами грамматики к какому–нибудь нетерминалу. Это можно сделать, если реализовать следующий алгоритм работы с магазином:

a) любой входной символ (кроме символа–ограничителя $\natural$ конца текста) записывается в магазин;

b) если в верхушке магазина сформирована основа, совпадающая с правой частью правила, то она заменяется на нетерминал в левой части этого правила;

c) разбор заканчивается, если в магазине аксиома, а входная цепочка просмотрена полностью (отсканирован маркер конца текста).

Недетерминированный характер работы такого алгоритма приводит к необходимости ограничить класс КС–грамматик так, чтобы можно было однозначно выбрать выполняемое действие. Один из таких методов основан на алгоритмах типа "перенос — свертка." Действительно, на каждом шаге восходящего анализа выполняется одно из двух действий:

1) очередной отсканированный элемент помещается в магазин — *перенос*;

2) верхушка магазина редуцируется к некоторому нетерминальному символу — *свертка.*

Таким образом, в магазине всегда находится левая часть текущей сентенциальной формы, состоящая из терминалов и нетерминалов, а правая ее часть всегда терминальна и формируется из отсканированной лексемы и еще не рассмотренной части исходного модуля.

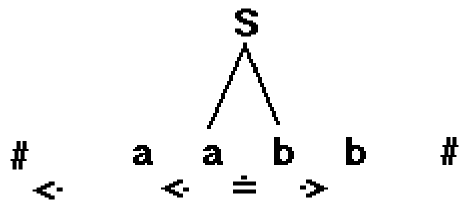
Рассмотрим ситуацию, когда верхушка магазина и отсканированная лексема однозначно определяют момент редукции. Для этого на множестве терминальных и нетерминальных символов введем отношения предшествования $< \cdot$, $\doteq$, $\cdot >$. Будем использовать эти отношения для выделения основы так, чтобы отношение $< \cdot$ выполнялось перед началом основы, отношение $\cdot >$ — на конце основы, а отношение

$\doteq$ между элементами основы. Для того, чтобы в начале и конце цепочки выполнять редукцию по тому же алгоритму, что и в ее середине, исходная цепочка дополняется слева и справа маркером конца, например, символом $\natural$. Очевидно, что отношения предшествования не рефлексивны (не всегда $a \doteq a$), не симметричны (если $a \doteq b$, то не обязательно $b \doteq a$), не транзитивны (если $a \doteq b$ и $b \doteq c$, то не обязательно $a \doteq c$), поскольку они определяют последовательность редукции и порядок элементов в сентенциальной форме, а эти элементы переставлять нельзя, не изменяя при этом язык.

Пример 8.1. В грамматике

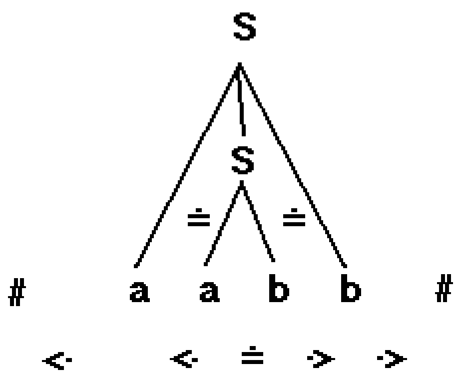
$$G : S \to aSb|ab$$

для цепочки $aabb$ должны выполняться отношения $\natural < \cdot a < \cdot a \doteq b \cdot > b\natural$, т.к. на первом шаге построения дерева разбора строится поддерево



Обратите внимание, что нет смысла рассматривать отношения после символа b, который следует за основой. Главная задача — определение момента редукции, и для этой цели достаточно найти первое отношение $\cdot >$.

На следующем шаге разбора в полученной цепочке $\natural aSb\natural$ должны выполняться отношения $\natural < \cdot a \doteq S \doteq b \cdot > \natural$, и дерево имеет вид:



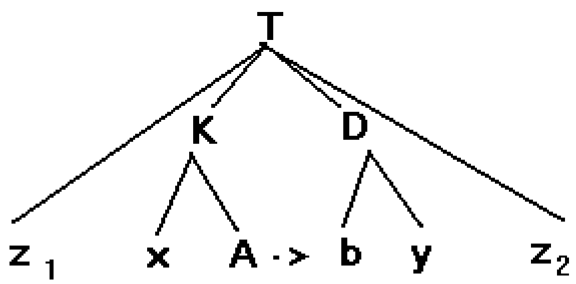
Это последний шаг разбора, т.к. получена аксиома S в окружении маркеров: $\natural S\natural$. В момент завершения разбора определять отношения между аксиомой и маркерами излишне.

8.1.2 Формальное определение отношений предшествования

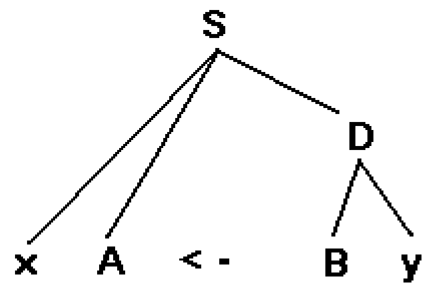
Итак, основу правовыводимой цепочки (см. стр. 204) можно выделить, просматривая эту цепочку слева направо до тех пор, пока впервые не встретится отношение $\cdot >$. Для нахождения левого конца основы надо возвращаться назад, пока не встретится отношение $< \cdot$. Цепочка, заключенная между $< \cdot$ и $\cdot >$, будет основой. Этот процесс продолжается до тех пор, пока либо не получим аксиому, либо из–за ошибок в тексте невозможны дальнейшие действия.

Определение 8.1. Для КС–грамматики $G = (V_T, V_N, P, S)$ на множестве $V_T \cup V_N \cup \{\natural\}$ установлены отношения предшествования следующим образом:

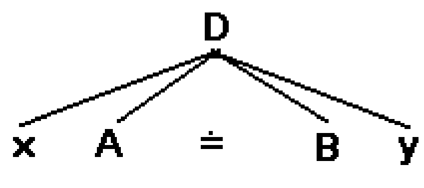
1) $A \cdot > b$, где $A \in V_T \cup V_N$, $b \in V_T$, если для некоторого нетерминала T имеется правило $T \to z_1 K D z_2$, существуют выводы $K \stackrel{+}{\Rightarrow} xA$ и $D \stackrel{*}{\Rightarrow} by$ (обратите внимание, что символ b рассматривается только из множества V_T, т.к. редукция осуществляется слева направо до тех пор, пока ее можно выполнить, и только при невозможности редукции можно прочитать следующий символ, следовательно непосредственно справа от основы может быть только терминальный символ); фрагмент дерева разбора в этом случае имеет вид



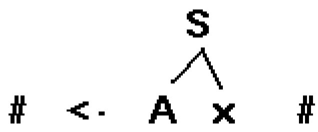
2) $A < \cdot B$, где $A, B \in V_T \cup V_N$, $S \stackrel{*}{\Rightarrow} xAD$, $D \stackrel{+}{\Rightarrow} By$ (символ A стоит перед основой, символ B начинает основу и, следовательно, редуцируется раньше символа A), фрагмент дерева разбора в этом случае имеет вид



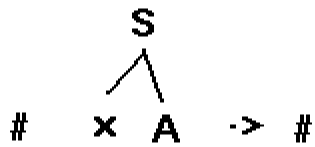
3) $A \doteq B$, где $A, B \in V_T \cup V_N$, если в множестве правил грамматики имеется правило $D \to xABy$, т.е. символы A и B принадлежат одной основе и, следовательно, редуцируются одновременно:



4) $\natural < \cdot A$, где $A \in V_T \cup V_N$, если $S \stackrel{+}{\Rightarrow} Ax$ (искусственно вставленные перед началом и после цепочки специальные маркеры $\natural$ используются для единообразия алгоритма определения первого и последнего символа основы как в середине цепочки, так и на ее концах), дерево разбора имеет вид



5) $A\cdot > \natural$, где $A \in V_T \cup V_N$, если $S \overset{+}{\Rightarrow} xA$, т.е. дерево разбора имеет вид



8.1.3 Матрица предшествования

Отношения предшествования удобно хранить в виде матрицы предшествования, строки и столбцы которой соответствуют символам из множества $V_T \cup V_N \cup \{\natural\}$.

Пример 8.2. Для грамматики

$$G: \quad \begin{array}{l} S \to SaA|A \\ A \to aAb|ab \end{array}$$

матрица предшествования имеет вид:

	S	A	a	b	$\natural$
S			$\doteq$		
A				$\doteq$	$\cdot >$
a		$\doteq$	$< \cdot$	$\doteq$	
b			$\cdot >$	$\cdot >$	$\cdot >$
$\natural$	$< \cdot$	$< \cdot$	$< \cdot$		

Пример грамматического разбора представлен на рис. 8.1.

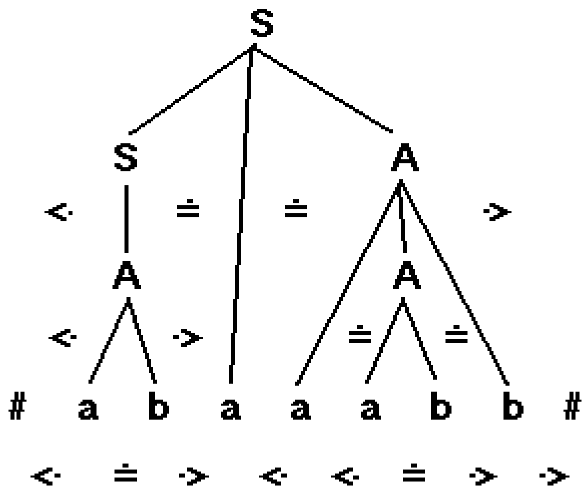


Рис. 8.1: Пример грамматического разбора в грамматике простого предшествования

Определение 8.2. Грамматика называется грамматикой предшествования, если между любыми двумя символами существует не более одного отношения предшествования.

Определение 8.3. Грамматика называется грамматикой простого предшествования, если она является грамматикой предшествования и все правила имеют различные правые части.

Рассмотрим на принципиальном уровне алгоритм работы анализатора в грамматике простого предшествования. Отношения $<\cdot$ и $\doteq$ вызывают необходимость перехода к следующему символу (сканирование очередной лексемы). Отношение $\cdot>$ приводит к выполнению редукции. В этот момент для выделения основы осуществляется возврат в магазине вниз от его верхушки по отношениям $\doteq$ до отношения $<\cdot$. В момент выделения основы — фрагмента сентенциальной формы между отношениями $<\cdot$ и $\cdot>$ — анализируются правила грамматики для поиска правила с выделенной правой частью. При редукции основа заменяется на нетерминал, а затем определяются отношения перед этим нетерминалом и после него.

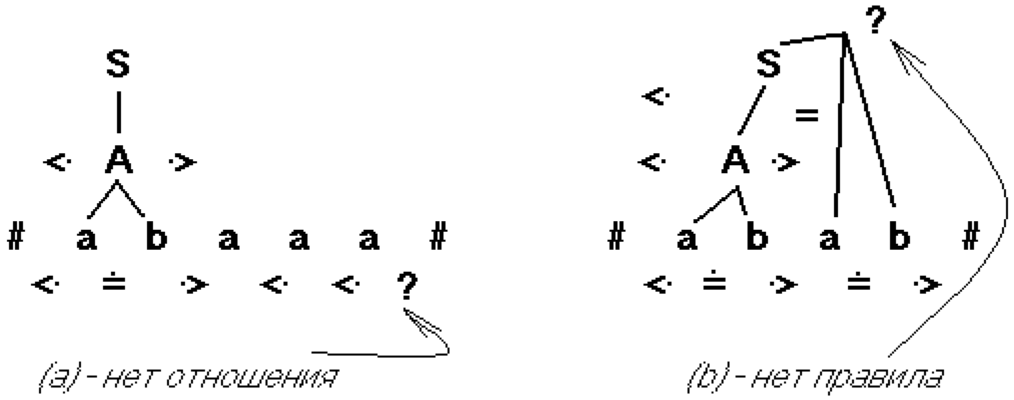


Рис. 8.2: Пример анализа ошибочных цепочек

Если в исходном модуле есть ошибки, то возможны два варианта их выявления. Во–первых, когда в процессе анализа смежными являются символы, для которых *нет отношений предшествования.* В грамматике из примера 8.2 такая ситуация возможна при анализе цепочки $abaaaa$ (рис. 8.2-a). Таким образом, если клетка матрицы пуста, то соответствующие символы не могут стоять рядом. Во-вторых, ошибка может быть выявлена в тот момент, когда *выделяется основа, которой не соответствует никакое правило грамматики.* В грамматике примера 8.2 эта ситуация появляется при анализе цепочки $abab$ (рис. 8.2-b).

8.1.4 Алгоритм построения матрицы предшествования

Определение 8.1 связывает отношения предшествования и выводимость смежных символов, поэтому для вычисления отношений предшествования построим множество N пар смежных символов, выводимых из цепочки $\natural S \natural$. Будем включать в это множество только такие пары символов, из которых хотя бы один является нетерминалом. Начальными элементами множества N являются пары $\natural S$ и $S\natural$. Остальные пары строятся по правилам грамматики. Затем из множества N сформируем два (возможно пересекающихся) подмножества. Пары вида Am, $A \in V_N$, $m \in V_T \cup \{\natural\}$, служат для построения отношений $\cdot >$. Для каждого правила грамматики $A \to ab...d$ записывается отношение $d\cdot > m$. Соответствующий фрагмент дерева разбора имеет вид, представленный на рис. 8.1.4(а).

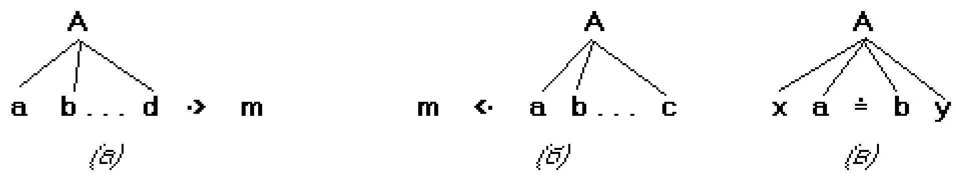


Рис. 8.3: Алгоритм построения отношений предшествования

Очевидно, что символ m может быть только терминальным, т.к. этот символ сканируется из входной цепочки. Если m рассматривался бы как нетерминал, то это означало бы построение дерева разбора беспорядочным способом, а не слева направо.

Пары вида mA, $A \in V_N$, $m \in V_T \cup V_N \cup \{\natural\}$, служат для построения отношений $< \cdot$. В соответствии с определением 8.1 для каждого правила грамматики $A \to ab...d$ необходимо записать отношение $m < \cdot a$. Соответствующий фрагмент дерева разбора имеет вид, представленный на рис. 8.1.4(б).

Указанный алгоритм строит все правильные отношения предшествования, но иногда среди построенных отношений встречаются лишние, которые не используются при синтаксическом анализе. Дело в том, что синтаксический анализ выполняется слева направо и редукции в некоторой точке выполняются до тех пор, пока это возможно. Рассмотрим некоторое правило $A \to \varphi a$, которое заканчивается терминальным символом a. Допустим, что терминал a во всех остальных правилах грамматики либо не встречается, либо встречается, но так же стоит последним в правой части, Тогда он может являться только концом основы и, следовательно, символ a может быть только больше некоторого символа, а отношение $a < \cdot b$ невозможно ни для какого b. Однако, если нетерминал A описывает фрагмент исходного модуля, который является не последним в программе, то в множестве пар обязательно появится пара AB, в которой символ B соответствует фрагменту, расположенному в исходном модуле за фрагментом A. Но тогда в множестве пар появится пара символов aB, т.к. существует вывод $AB \Rightarrow \varphi aB$. Следовательно, рассмотренный выше алгоритм построит отношение $a < \cdot first_1(B)$, конфликтующее с отношением $a\cdot > first_1(B)$.

Эти соображения приводят к необходимости дополнения указанного алгоритмя некоторыми ограничениями. Для того, чтобы из матрицы предшествования исключить лишние отношения, из всех пар вида mA вычеркиваем пары с первым символом m, который во всех правилах грамматики, его содержащих, встречается только в качестве последнего символа правой части. Это означает, что символ m должен быть редуцирован раньше, чем та часть исходного модуля, которая сводится к A. Только после такого удаления лишних пар формируем отношения $< \cdot$.

Вообще говоря, указанное удаление пар можно и не делать. Дело в том, что в данной ситуации построенные отношения конфликтуют: $a < \cdot first_1(B)$ и $a\cdot > first_1(B)$. Наличие конфликта приводит к необходимости его анализа и выбору методов устранения. В данном случае работает один из двух вариантов действий: либо сразу не вносить в таблицу фактически не имеющие места конфликты, либо потом устранить их.

В конце для всех правил $A \to xaby$, правая часть которых содержит не менее двух символов, записываем отношение $a \doteq b$ (см. рис. 8.1.4(в)).

Сразу отметим, что наличие укорачивающих правил однозначно свидетельствует о том, что метод предшествования для грамматики применять невозможно из-за наличия конфликтов. Действительно, если между символами a и b должна выполняться редукция по правилу с пустой правой частью, то должны выполняться два отношения:

1) $a <\cdot b$, т.к. после символа a находится начало основы;

2) $a \cdot> b$, т.к. перед символом b находится конец основы.

Пример 8.3. Построим отношения предшествования для грамматики

$$\begin{aligned} G:\ & S \to SaA|A \\ & A \to aAb|ab. \end{aligned}$$

$N = \{\natural S, S\natural, Sa, aA, \natural A, A\natural, Aa, Ab\}$. Тогда для построения отношения $\cdot >$ используются следующие пары:

$$\begin{aligned} S\natural:\ & A\cdot > \natural, \\ Sa:\ & A\cdot > a, \\ A\natural:\ & b\cdot > \natural, \\ Aa:\ & b\cdot > a, \\ Ab:\ & b\cdot > b. \end{aligned}$$

Для построения $< \cdot$ — пары:

$$\begin{aligned} \natural S:\ & \natural < \cdot S, \natural < \cdot A, \\ aA:\ & a < \cdot a, \\ \natural A:\ & \natural < \cdot a. \end{aligned}$$

Символ b во всех правилах стоит только последним, поэтому, если бы какая–нибудь из построенных нами пар смежных символов начиналась бы символом b, то все такие пары надо было бы выбросить.

И, наконец, по правилам грамматики определяем отношения равенства:

$$\begin{aligned} S \to SaA:\ & S \doteq a, a \doteq A, \\ A \to aAb:\ & a \doteq A, A \doteq b, \\ A \to ab:\ & a \doteq b. \end{aligned}$$

8.2 Функции предшествования

Как и любой другой основанный на таблицах алгоритм, метод предшествования может быть реализован в виде программ двух типов:

1) универсальной программы, в которой используется матрица и выполняется поиск по этой матрице (явный способ);

2) программы, фрагменты которой встроены в виде операторов, соответствующих элементам матрицы предшествования (неявный способ).

При использовании явного метода матрица занимает в памяти $n \times n$ элементов. Рассмотрим условия сокращения памяти на порядок: $O(n^2) \to O(2n)$.

Определение 8.4. Функции предшествования для матрицы предшествования $M[n, n]$ — это пара целочисленных векторов $f[n]$, $g[n]$, таких, что

a) если $M[i, j] =< \cdot$, то $f[i] < g[j]$;

b) если $M[i, j] = \cdot >$, то $f[i] > g[j]$;

c) если $M[i, j] = \doteq$, то $f[i] = g[j]$.

Сразу следует отметить, что поскольку между любыми двумя числами всегда имеется какое-то отношение, то при использовании функций предшествования теряется возможность обнаружения ошибки по пустому элементу матрицы предшествования. Остается только один вариант обнаружения ошибки — по отсутствию среди правил грамматики такого правила, правая часть которого совпадает с выделенной основой.

Пример 8.4. Для грамматики

$$G: \begin{array}{l} S \to SaA|A \\ A \to bAc|bc \end{array}$$

функции предшествования равны

$$\begin{array}{l} f = (0, 1, 1, 1, 2, 0), \\ g = (1, 1, 0, 2, 1, 0), \end{array}$$

где элементы векторов f и g соответствуют последовательности символов $\{S,\ A,\ a,\ b,\ c,\ \natural\}$.

Восстановленная по функциям матрица предшествования не содержит пустых клеток и имеет вид:

	1	1	0	2	1	0	
0	$<\cdot$	$<\cdot$	$\doteq$	$<\cdot$	$<\cdot$	$\doteq$	S
1	$\doteq$	$\doteq$	$\cdot>$	$<\cdot$	$\doteq$	$\cdot>$	A
1	$\doteq$	$\doteq$	$\cdot>$	$<\cdot$	$\doteq$	$\cdot>$	a
1	$\doteq$	$\doteq$	$\cdot>$	$<\cdot$	$\doteq$	$\cdot>$	b
2	$\cdot>$	$\cdot>$	$\cdot>$	$\doteq$	$\cdot>$	$\cdot>$	c
0	$<\cdot$	$<\cdot$	$\doteq$	$<\cdot$	$<\cdot$	$\doteq$	$\natural$
	S	A	a	b	c	$\natural$	

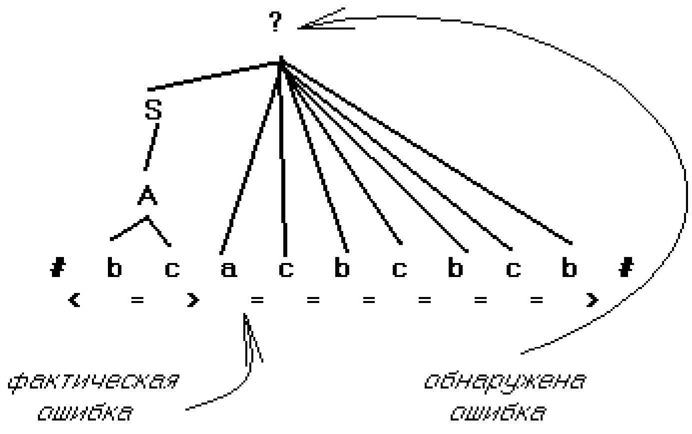


Рис. 8.4: Выявление ошибок с помощью функций предшестповвания

Рассмотрим разбор неверной цепочки, представленный на рис. 8.4. Отметим, что функции предшествования обнаруживают ошибку с возможным смещением вправо, поэтому в дальнейшем рассмотрим способы, позволяющие внести возможность кодирования ошибки с помощью функций некоторого расширенного вида. А пока перейдем к алгоритму построения функций предшествования. Пусть матрице предшествования поставлен в соответствие граф с вершинами двух типов: f и g. Вершины типа f будут соответствовать строкам матрицы, а вершины типа g — столбцам. Будем строить граф так, чтобы в качестве элементов векторов f и g была выбрана максимальная длина пути, выходящего из данной вершины. Для того, чтобы выполнялось это условие, рассмотрим последовательно все отношения в матрице. Для того, чтобы обеспечить отношение $\doteq$, все вершины f и g, соответствующие этому отношению, объединим в одну вершину. Максимальные длины путей, выходящих из f и g, в этом случае с необходимостью совпадут. Отношение $\cdot >$ между f и g должно быть реализовано так, чтобы путь из f был длиннее пути из g. Для этого достаточно провести дугу из f в g. Аналогично отношению $< \cdot$ должна соответствовать дуга обратного направления из g в f. Построенный граф называется *графом линеаризации*. Для каждой вершины графа линеаризации найдем длину максимального выходящего пути и построим соответствующие вектора f и g. Очевидно, что наличие циклов в графе означает невозможность построения для данной матрицы функций предшествования. Для программной реализации синтаксического анализатора, основанного на функциях предшествования, в таком случае можно убрать одно отношение из цикла, построить функции предшествования, а потом это отношение запрограммировать отдельно.

Пример 8.5. Построить функции предшествования для грамматики из примера 8.4, матрица предшествования которой имеет вид

	1	2	3	4	5	6
1			$\doteq$			
2			$\cdot >$		$\doteq$	$\cdot >$
3		$\doteq$		$< \cdot$		
4		$\doteq$		$< \cdot$	$\doteq$	
5			$\cdot >$		$\cdot >$	$\cdot >$
6	$< \cdot$	$< \cdot$		$< \cdot$		

Сливаем вершины

a) f[1], g[3];

b) f[2], g[5], f[4], g[2], f[3].

Строим граф линеаризации, представленный на рис. 8.5. Вычислим максимальные длины выходящих путей и получим $f = (0, 1, 1, 1, 2, 0)$, $g = (1, 1, 0, 2, 1, 0)$.

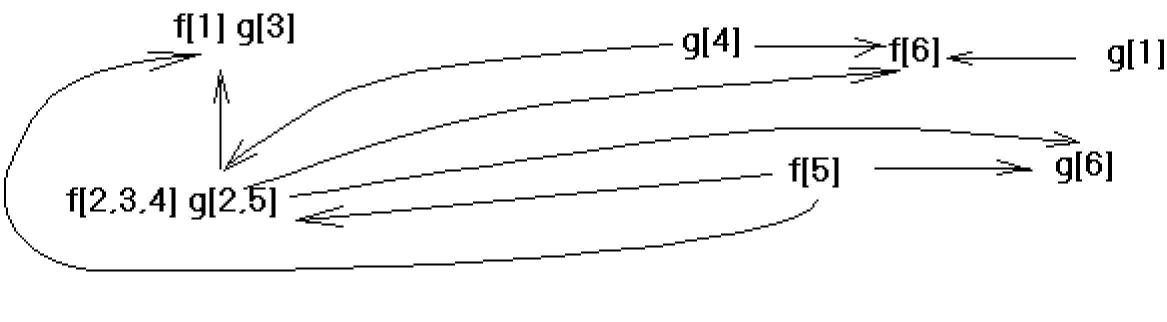


Рис. 8.5: Граф линеаризации

В программе синтаксического анализатора, который запрограммирован *явным способом*, для вычисления отношений в данной грамматике достаточно тепреь использовать два вектора f и g, а отношение между символами с типами i и j вычислять как отношение между $f[i]$ и $g[j]$.

8.3 Отношения и функции слабого предшествования

8.3.1 Определение отношений слабого предшествования

Рассмотрим способ представления матрицы предшествования с помощью некоторых функций предшествования так, чтобы имелась возможность кодирования ошибок — пустых клеток матрицы. В матрице предшествования содержатся элементы четырех типов: три типа отношений ($\cdot >$, $< \cdot$, $\doteq$) и пустые клетки. Между любыми двумя числами можно установить только три отношения. Следовательно, требуется представить какие–либо два отношения одним типом. Отношения $< \cdot$ и $\doteq$ вызывают одинаковые действия по сканированию очередного символа и записи предшествующего символа в магазин. Различие отношений $< \cdot$ и $\doteq$ сказывается в момент редукции, т.к. для выделения основы нужно возвращаться по магазину по отношениям $\doteq$ до отношения $< \cdot$. Найдем другой способ определения основы, а отношения $< \cdot$ и $\doteq$ соединим в одно отношение $\leq \cdot$. Воспользуемся следующим фактом: если все правила грамматики имеют разные окончания, то основу можно определить с помощью сравнения верхушки магазина с этими правилами. Возникает вопрос: а нельзя ли воспользоваться эти методом определения применяемого правила и в общем случае? Пусть в грамматике есть два правила, одно из которых является окончанием другого:

$$\begin{array}{l} A \to x \\ B \to yx. \end{array} \tag{8.1}$$

Определим условия, при которых редукция осуществляется по самому длинному подходящему правилу. Допустим, в грамматике редукция допустима как по длинному, так и по короткому правилу. Тогда должно существовать отношение $< \cdot$ или $\doteq$ между $last(y)$ и символом A, что эквивалентно существованию двух правильных деревьев разбора:

Обратно, если не существуют указанные отношения между $last(y)$ и A, тогда невозможна редукция по любому из правил (8.1), следовательно, она выполняется по единственному самому длинному правилу.

Определение 8.5. Грамматика называется грамматикой слабого предшествования, если

1) между любыми двумя символами существует не более одного отношения $\leq \cdot$ или $\cdot >$;

2) для правил типа (8.1) не существует отношения между $last(y)$ и A.

Отношения слабого предшествования позволяют, во-первых, избавиться от конфликтов $< \cdot$ и $\doteq$, т.к. эти два отношения сливаются. Во-вторых, в матрице слабого предшествования имеются только три типа элементов, что позволяет в дальнейшем кодировать ошибки с помощью функций слабого предшествования.

8.3.2 Построение функций слабого предшествования

Если отношения слабого предшествования — это отношения $\leq\cdot$ и $\cdot>$, тогда для кодирования ошибок с помощью функций можно использовать отношение равенства между соответствующими элементами векторов f и g.

Определение 8.6. Функциями слабого предшествования для матрицы слабого предшествования M называется такая пара целочисленных векторов f и g, что элементу матрицы $M[i,j] =\leq\cdot$ соответствует отношение между числами $f[i] < g[j]$, элементу $M[i,j] = \cdot>$ — отношение $f[i] > g[j]$, а отношению $f[i] = g[j]$ соответствует $M[i,j] =$ "ошибка".

Заметим, что третье условие нельзя сформулировать в обратном порядке, т.к. не каждой матрице предшествования соответствуют функции предшествования, поэтому попытка определить любую ошибку матрицы M через отношение равенства на f и g может привести к невозможности построения функций. Поскольку попытка определить любую ошибку через функции может быть неудачной, будем строить такие функции слабого предшествования для матрицы слабого предшествования, которые определяют максимальное число выявляемых ошибок.

Для обеспечения отношения $f[i] = g[j]$ достаточно слить соответствующие вершины графа линеаризации, а для кодирования максимального числа ошибок необходимо слить максимальное число таких вершин. Чтобы в полученном графе не возникли циклы, такому слиянию можно подвергнуть только независимые вершины, т.е. вершины, между которыми нет пути. Причем, чтобы обеспечить максимальное число выявляемых ошибок, требуется сливать вершины такого множества независимых вершин, которое дает максимальное число попарного сочетания $f[i]$ и $g[j]$.

Пример 8.6. Для грамматики

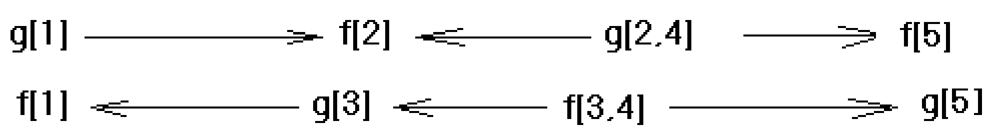
$$G : S \to aSb|c$$

матрица предшествования и матрица слабого предшествования имеют вид:

	S	a	b	c	$\natural$
S			$\doteq$		
a	$\doteq$	$<\cdot$		$<\cdot$	
b			$\cdot>$		$\cdot>$
c			$\cdot>$		$\cdot>$
$\natural$		$<\cdot$		$<\cdot$	

	S	a	b	c	$\natural$
S			$\leq\cdot$		
a	$\leq\cdot$	$\leq\cdot$		$\leq\cdot$	
b			$\cdot>$		$\cdot>$
c			$\cdot>$		$\cdot>$
$\natural$		$\leq\cdot$		$\leq\cdot$	

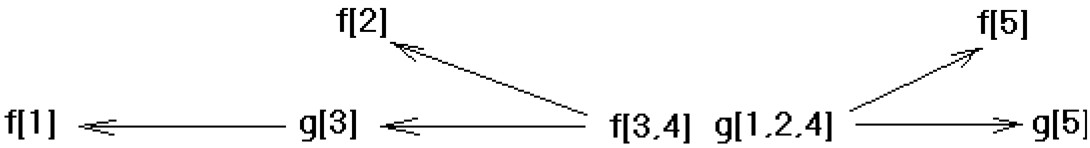
Одинаковые строки и одинаковые столбцы в матрице можно слить, т.к. преобразования графа для этих элементов в дальнейшем будут осуществляться одинаково. Строим граф линеаризации по обычным правилам:



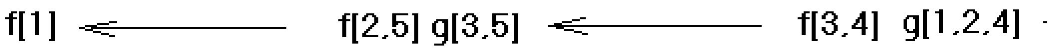
Находим множество независимых вершин с максимальным числом сочетаний пар f и g. Таким множеством является

$$f[3,4],\ \ g[1,2,4]$$

с числом сочетаний 2*3=6 (например, для множества $f[1]$, $g[1,2,4,5]$ число пар меньше 6 и равно 4, а множество $f[1,2,3,4,5]$ вообще нет смысла сливать, т.к. при этом число выявляемых ошибок будет равно нулю). В результате слияния вершин получим граф:



Повторяем алгоритм для полученного графа. Получим граф линеаризации, в котором все вершины зависимы:



Для построенного графа определяем значения элементов векторов:

$$f = (0,1,2,2,1), g = (2,2,1,2,1).$$

Матрица предшествования, восстановленная по функциям, имеет вид:

	2	2	1	2	1
0	$\le\cdot$	$\le\cdot$	$\le\cdot$	$\le\cdot$	$\le\cdot$
1	$\le\cdot$	$\le\cdot$	ош	$\le\cdot$	ош
2	ош	ош	$\cdot>$	ош	$\cdot>$
2	ош	ош	$\cdot>$	ош	$\cdot>$
1	$\le\cdot$	$\le\cdot$	ош	$\le\cdot$	ош

Таким образом, построенные функции слабого предшествования из 15 ошибок выявляют 10.

8.4 Преобразование КС–грамматики к грамматике предшествования

Рассмотрим удаление конфликтов в матрице предшествования. Если в матрице наблюдаются только конфликты $<\cdot$ и $\doteq$, можно попробовать перейти к отношениям слабого предшествования, проверив предварительно грамматику на условия слабого предшествования. Если эти условия не удовлетворяются или конфликты имеют другую природу, то можно попробовать преобразовать грамматику к грамматике предшествования. Следующая теорема показывает, что такое преобразование всегда возможно и предлагает алгоритм этого преобразования. Правда, следует отметить, что полученная грамматика не обязательно является грамматикой простого предшествования. Более того, применение алгоритма преобразования в полной форме приведет к большому числу одинаковых правил. Поэтому предлагаемый алгоритм обычно применяют в урезанной форме только для тех правил, из-за которых и возникли конфликты отношений предшествования.

Теорема 8.1. Для любой КС–грамматики существует эквивалентная грамматика предшествования.

Доказательство. Пусть имеется КС-грамматика $G = (V_T, V_N, P, S)$. Так как любую КС–грамматику можно преобразовать к эквивалентной неукорачивающей, можем считать, что G — неукорачивающая грамматика. Рассмотрим правила, в правой части которых более одного символа:

$$A \to a_1 a_2 ... a_k,\ \ k \geq 1.$$

Заменим каждое такое правило на эквивалентные правила

$$\begin{array}{ll} A \to & B_1 B_2 ... B_k \\ B_1 \to & a_1 \\ B_2 \to & a_2 \\ & ... \\ B_k \to & a_k, \end{array}$$

где B_i — новые нетерминалы, уникальные для каждого правила. Покажем, что построенная грамматика является грамматикой предшествования. Для этого достаточно рассмотреть пары смежных символов и показать, что между любыми символами в таких парах существует не более одного отношения предшествования. Имеется четыре варианта смежных пар "новых" и "старых" символов: SN, NN, NS, SS, где N — некоторый "новый", а S — некоторый "старый" символ.

Рассмотрим, например, пару символов SN. Пусть $S \doteq N$, тогда существует правило $A \to xSNy$, что невозможно по построению. Отношение $S < \cdot N$ также невозможно, поскольку все "старые" символы являются единственными в правой части правил и подлежат редукции. Остается единственно возможное отношение $S \cdot > N$.

Рассмотрим теперь пару символов $N_1 N_2$. Допустим, $N_1 \doteq N_2$, тогда должно существовать правило $D \to xN_1N_2y$. Отметим, что в соответствии с принципом уникальности новых нетерминалов это правило является единственным с нетерминалами N_1 и N_2 в правой части. Пусть одновременно с отношением $N_1 \doteq N_2$ выполняется отношение $N_1 < \cdot N_2$, но тогда символ N_2 является началом основы, а в единственном правиле $D \to xN_1N_2y$ символ N_2 занимает не самую левую позицию. Пусть одновременно с отношением $N_1 \doteq N_2$ выполняется отношение $N_1 \cdot > N_2$, но тогда символ N_1 является концом основы, а в единственном правиле, содержащем N_1, символ N_1 занимает не самую правую позицию.

Пусть теперь $N_1 < \cdot N_2$, тогда по определению отношения $< \cdot$ существует правило $D \to xN_1Ty$ и вывод $T \overset{+}{\Rightarrow} N_2 z$, а из доказанного выше следует невозможность существования отношения $N_1 < \cdot N_2$ с одновременным существованием отношения $N_1 \doteq N_2$. Пусть одновременно с $N_1 < \cdot N_2$ выполняется $N_1 \cdot > N_2$, но тогда N_1 является концом основы и должен стоять последним в правой части некоторого правила, что противоречит единственности правила $D \to xN_1Ty$.

Отношение $N_1 \cdot > N_2$ из доказанного выше не может существовать одновременно с отношениями $N_1 < \cdot N_2$ и $N_1 \doteq N_2$.

Остальные пары символов рассматриваются аналогично. $\square$

Построенный алгоритм преобразования приводит к громоздкой грамматике, в которой очень много нетерминалов и правил с одинаковыми правыми частями. Как уже отмечалось, на практике данный алгоритм применяют к тем смежным символам, из–за которых возникли конфликты, и вводят двойников только в точке конфликта. Более того, этот алгоритм применяют только при сложных конфликтах $(< \cdot, \cdot >)$ или $(\leq \cdot, \cdot >)$, а при конфликтах $\leq \cdot$ или $\cdot \geq$ применяют более простые методы преобразования.

Рассмотрим конфликт $\leq \cdot$. Пусть $a \doteq b$ и $a < \cdot b$, тогда есть правило $A \to xaby$. Заменим это правило на два эквивалентных правила $A \to xaT$ и $T \to by$. В результате поддерево разбора для нетерминала A будет соответствовать выводу $A \Rightarrow xaT \Rightarrow xaby$. Отношение $\doteq$ между символами a и b заменено отношением $< \cdot$. Конфликт исчез.

Аналогично конфликт $\cdot \geq$ между символами a и b возникает только при наличии правила $A \rightarrow xaby$. Заменив это правило на два эквивалентных $A \rightarrow Tby$ и $T \rightarrow xa$, получим отношение $a \cdot > b$ и удалим конфликт.

Указанные алгоритмы устранения конфликтов не требуют повторного построения матрицы предшествования, т.к. достаточно только дополнить матрицу новой строкой и столбцом для нового нетерминала T, перенести в строку (столбец) отношение равенства и продублировать отношения из строки (столбца) для символа a (соответственно для символа b).

8.5 Расширенное предшествование

Другой метод устранения конфликтов основан на расширении контекста. Рассмотрим две грамматики

$$G_1: \quad S \rightarrow aSbb|bb|cbb$$
$$G_2: \quad S \rightarrow aaSb|aa|aac.$$

Обе эти грамматики не являются грамматиками простого предшествования, т.к. в G_1 существуют отношения $b \doteq b$ и $b \cdot > b$, а в грамматике G_2 имеются отношения $a \doteq a$ и $a < \cdot a$. Этот факт хорошо иллюстрируют представленные на рис 8.6 примеры деревьев разбора в этих грамматиках.

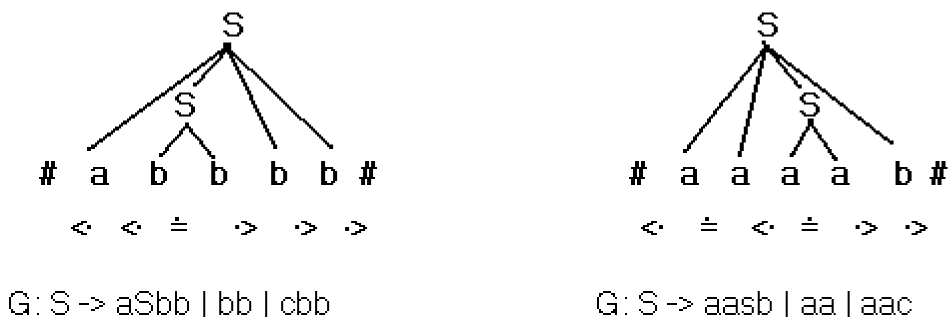


Рис. 8.6: Расширенный контекст вправо и влево.

В грамматике G_1 вывод относительно отношений $\doteq$ или $\cdot >$ между символами bb можно сделать на основе анализа расширенного контекста вправо, выполняя анализ двух символов справа вместо одного. В грамматике G_2 вывод относительно отношений $\doteq$ или $< \cdot$ между aa можно сделать на основе анализа расширенного контекста влево. Эти примеры показывают, что можно рассмотреть отношения предшествования не между символами, а между цепочками. Такие отношения называются отношениями (m, n)–предшествования между цепочками x и y, $|x| = m$, $|y| = n$.

Определение 8.7. Для КС-грамматики $G = (V_T, V_N, P, S)$ между цепочками x и y ($x, y \in (V_T \cup V_N \cup \{\natural\})^*$, $|x| = m$, $|y| = n$) установлены отношения расширенного (m, n)–предшествования следующим образом:

1) $x < \cdot y$, если $\natural^m S \natural^n \overset{*}{\Rightarrow} wxBz$, $B \overset{+}{\Rightarrow} u$, $y \in First_n(uz)$, где функция $First_n(v)$ отличается от обычной функции $first_n(v)$ тем, что определяет начала выводимых из v цепочек, состоящих не только из терминальных символов;

2) $x \cdot > y$, если $\natural^m S \natural^n \stackrel{*}{\Rightarrow} wByz$, $B \stackrel{+}{\Rightarrow} u$, $x \in Last_m(wu)$; отличия функции $Last_m(v)$ от функции $last_m(v)$ аналогичны отличиям функции $First_m(v)$ от $first_m(v)$;

3) $x \doteq y$, если в грамматике G существует правило $A \to uabv$ и вывод $\natural^m S \natural^n \stackrel{*}{\Rightarrow} wAz$, где $x \in Last_m(wua)$ и $y \in First_n(bvz)$.

Определение может быть проиллюстрировано поддеревьями, соответствующими указанным отношениям между цепочками (см. рис. 8.7).

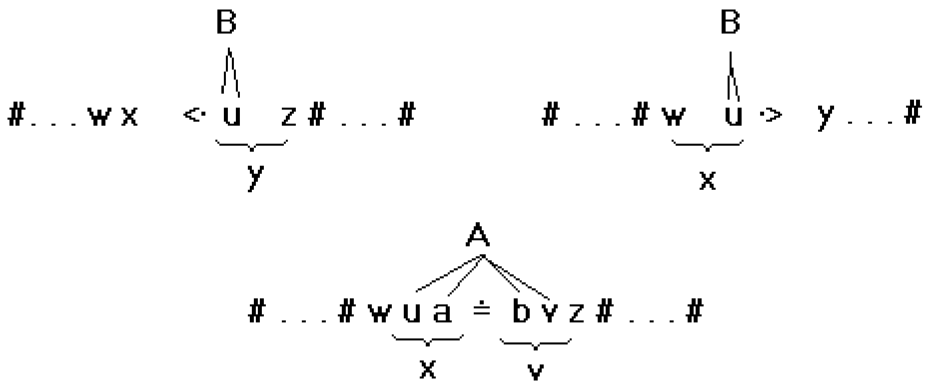


Рис. 8.7: Определение расширенного предшествования

Отношения (m, n)–предшествования имеют частный случай — (1,1)-предшествование, совпадающее с простым предшествованием. В соответствии с определением расширенного предшествования можно получить алгоритм построения этих отношений обобщением алгоритма построения отношений простого предшествования. Если (1,1)–предшествование было основано на анализе смежных пар символов, выводимых из цепочки $\natural S \natural$, то аналогичный алгоритм для расширенного предшествования должен быть основан на формировании множества цепочек длины m+n, выводимых из $\natural^m S \natural^n$. Рассмотрим работу алгоритма на примере.

Пример 8.7. Построить отношения (2,1)–предшествования для грамматики

$$G: \quad S \to aSbb|bb|cbb.$$

Сумма длин $m + n$ равна трем, поэтому строим множество цепочек длины 3, выводимых из $\natural\natural S \natural$:

$$N = \{\natural\natural S, \natural S \natural, \natural a S, aSb, Sbb, aaS\}.$$

Элементы этого множества получены из выводов

$$\begin{array}{ll} \natural\natural S \stackrel{*}{\Rightarrow} & \natural\natural aSbb|\natural\natural bb|\natural cbb, \\ \natural a S \stackrel{*}{\Rightarrow} \natural aaSbb. & \end{array}$$

Остальные возможные выводы не дадут новых цепочек длины 3, содержащих хотя бы один нетерминал.

Цепочки $\natural\natural S$, $\natural a S$, aaS дают отношения

$$\{\natural\natural, \natural a, aa\} < \cdot \{a, b, c\},$$

а цепочки $\natural S \natural$, aSb — отношения

$$\{bb\} \cdot > \{\natural, b\}.$$

Для построения отношений равенства достаточно рассмотреть цепочки $\natural\natural S$ и $\natural a S$, т.к. остальные цепочки множества N не дают нового контекста по сравнению с ними:

$$\begin{array}{c} \natural\natural a \doteq S \doteq b \doteq b \mid \natural\natural b \doteq b \mid \natural\natural c \doteq b \doteq b, \\ \natural aa \doteq S \doteq b \doteq b \mid \natural ab \doteq b \mid \natural aa \doteq b \doteq b. \end{array}$$

Построенные отношения занесем в таблицу, строки которой соответствуют цепочкам длины 2, а столбцы - цепочкам длины 1:

	S	a	b	c	$\natural$
$\natural a$	$\doteq$	$<\cdot$	$<\cdot$	$<\cdot$	
aS			$\doteq$		
Sb			$\doteq$		
$\natural b$			$\doteq$		
aa	$\doteq$	$<\cdot$	$<\cdot$	$<\cdot$	
ac			$\doteq$		
$\natural c$			$\doteq$		
cb			$\doteq$		
ab			$\doteq$		
bb			$\cdot>$		$\cdot>$
$\natural\natural$		$<\cdot$	$<\cdot$	$<\cdot$	

Очевидно, что расширенное предшествование основано на матрице больших размеров: если простое предшествование использует матрицу размера $O(N^2)$, то расширенное — размера $O(N^{m+n})$. Это означает принципиальный рост объема данных или величины программы. Конечно, такие характеристики программы совершенно неприемлемы. Но важна сама идея расширенного предшествования — определить отношение, скажем, между парами смежных символов. Фактически, такую проверку нужно делать не всегда, а только в том случае, когда простое предшествование не позволяет однозначно определить действие анализатора. Это значит, что указанную проверку нужно выполнять только в случае возникновения конфликта в матрице простого предшествования.

8.6 Операторное предшествование

Обычно для перевода исходного модуля в объектный код достаточно иметь остов дерева разбора. Рассмотрим остовы деревьев разбора двух выражений, различающихся двумя переставленными знаками операций (см. рис. 8.8).

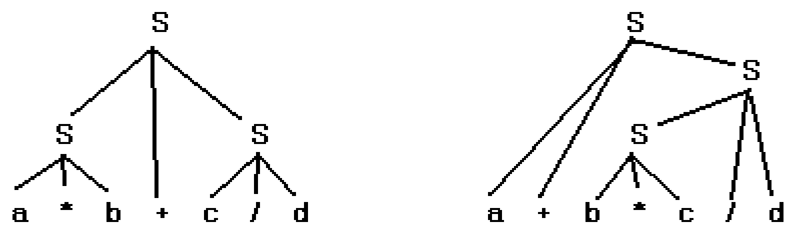


Рис. 8.8: Остовы деревьев разбора арифметических выражений

Пример показывает следующие характерные особенности разбора:

1) соседние терминальные элементы входной цепочки определяют последовательность редукции;

2) при редукции получаем нетерминал, который делает прозрачной соответствующую часть входной цепочки.

Поэтому будем рассматривать отношения предшествования только на множестве терминалов, считая при этом, как и всегда, ограничитель начала и конца исходного модуля тоже терминальным символом. В этом случае все нетерминалы сливаются в один нетерминал в силу их прозрачности — в качестве такого нетерминала можно выбрать аксиому грамматики. Предлагаемый подход повышает скорость разбора, т.к. снижается высота дерева и не требуется анализ нетерминала. Однако, такой подход возможен для грамматики с некоторыми ограничениями на правила вывода. Допустим, имеются правила:

$$D \to xABy,$$
$$A \to wT,$$
$$B \to Fu,$$

тогда возможен вывод $D \Rightarrow xABy \overset{+}{\Rightarrow} xwTFuy$, в результате которого появились смежные терминалы w и u (с учетом прозрачности нетерминалов), слабосвязанные контекстом. Поэтому вводится следующее определение.

Определение 8.8. Грамматика называется операторной, если она неукорачивающая и в правых частях правил отсутствуют смежные нетерминалы.

Определение 8.9. Для операторной грамматики $G = (V_T, V_N, P, S)$ на множестве терминальных символов устанавливаются отношения операторного предшествования:

1) $\natural < \cdot$ a, если $S \overset{+}{\Rightarrow} Taw$, где $T \in V_N \cup \{\varepsilon\}$;

2) $a\cdot > \natural$, если $S \overset{+}{\Rightarrow} waT$;

3) $a \doteq b$, если в P имеется правило $A \to waTbu$;

4) $a\cdot > b$, если $S \overset{*}{\Rightarrow} wAbu$ и $A \overset{+}{\Rightarrow} yaT$;

5) $a < \cdot b$, если $S \overset{*}{\Rightarrow} waAu$ и $A \overset{+}{\Rightarrow} Tby$.

Определение 8.10. Операторная грамматика называется грамматикой операторного предшествования, если между любыми двумя терминальными символами выполняется не более одного отношения операторного предшествования.

Алгоритм построений отношения операторного предшествования практически совпадает с алгоритмом построения отношений простого предшествования за тем исключением, что в процессе определения отношений нетерминалы прозрачны.

Пример 8.8. Построить отношения операторного предшествования для грамматики

$$G : S \to S + A|S - A|A$$
$$A \to A * B|A/B|B$$
$$B \to c|a|(S)$$

Строим множество пар смежных символов, из которых один терминал, а второй - нетерминал:

$$N = \{S\natural, \natural S, S+, S-, A*, A/, +A, -A, A\natural, (S, S), \natural A, A\natural, *B, /B, A+, A-, B\natural, \natural B, +B, -B, (A, A), (B, B), B+, B-, B*, B/\}.$$

Строим отношение $\cdot >$, выбирая пары типа "нетерминал — терминал", получим:

для $S\{\natural, +, -,)\}$ — отношения $\{+, -\}\cdot > \{\natural, +, -,)\}$,

для $A\{\natural, *, /, +, -,)\}$ — отношения $\{*, /\}\cdot > \{\natural, *, /, +, -,)\}$,

для $B\{\natural, +, -, *, /,)\}$ — отношения $\{a, c,)\}\cdot > \{\natural, +, -, *, /)\}$.

Строим отношение $< \cdot$, выбирая пары типа "терминал — нетерминал" и исключая те из них, которые заканчиваются последним символом во всех правилах грамматики (если, конечно, такие пары имеются в построенном множестве), получим:

для $\{\natural, (\}S$ — отношения $\{\natural, (\} < \cdot\{+, -\}$,

для $\{\natural, (, +.-\}A$ — отношения $\{\natural, (, +, -\} < \cdot\{*, /\}$,

для $\{\natural, (, *, /, +, -\}B$ — отношения $\{\natural, (, *, /, +, -\} < \cdot\{a, c, (\}$.

Отношение "$\doteq$" строим по правилам грамматики: $\{(\} = \{)\}$. Отношения занесем в матрицу операторного предшествования:

	+-	*/	c	a	(	)	$\natural$
+-	$(>$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
*/	$\cdot>$	$\cdot>$	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
c	$\cdot>$	$\cdot>$				$\cdot>$	$\cdot>$
a	$\cdot>$	$\cdot>$				$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$\doteq$	
)	$\cdot>$	$\cdot>$				$\cdot>$	$\cdot>$
$\natural$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$	$<\cdot$		

8.7 Реализация анализаторов предшествования

Анализатор предшествования имеет особенности, связанные с типом используемых отношений предшествования, однако, эти особенности оказывают влияние на реализацию только некоторых фрагментов программы и не изменяют алгоритма в целом. Алгоритм анализатора предшествования представлен на рис. 8.9. Рассмотрим особенности реализации некоторых блоков, изображенных на этом рисунке (здесь, как и ранее, Mag — магазин, Lex — отсканированная лексема, представленная типом или изображением).

При выполнении блока (1) возможно выполнение некоторой семантической подпрограммы одновременно с записью лексемы Lex в магазин Mag.

Реализация блока (2) имеет особенности для операторного предшествования. Так как все нетерминалы неразличимы, то редукция к нетерминалу реализуется просто записью признака нетерминала в магазин Mag. Чаще всего правила грамматики операторного предшествования достаточно просты и таковы, что терминальные символы в правых частях однозначно определяют применяемые правила. Тогда возможны варианты реализации без записи нетерминала в магазин. Можно предложить следующие способы обработки нетерминалов для операторного предшествования.

а) В простейшем случае нетерминалы в магазин не пишутся вообще, магазин предназначен только для терминалов; следует учесть, что в таком случае возникают некоторые трудности при выявлении ошибок, соответствующих пропущенным элементам (например, при трансляции выражения $a + + * +a$);

б) Иногда правила грамматики различаются по нетерминалу в конце правой части, например, при использовании одинакового знака для унарных и бинарных операций ($S \rightarrow +A| - A|S + A|S - A|A$). Тогда для того, чтобы при необходимости различить их при редукции, нетерминал записывается не в верхушку магазина Mag, а над ней. В этом случае на последующем шаге при редукции анализируется элемент над верхушкой магазина, а при занесении нового символа в магазин он просто затирается этим новым символом.

в) Наиболее надежным и простым способом реализации редукции по любому правилу является запись в магазин нетерминала, соответствующего аксиоме.

Блок (3) — это либо поиск отношения в матрице, либо вычисление отношения через функции предшествования, либо с помощью операторов условия, встроенных в программу. При выделении основы в блоке (4) при использовании простого предшествования или простого операторного достаточно организовать просмотр верхушки Mag вниз по отношениям "$\doteq$" до отношения " $<\cdot$ ". Любое слабое предшествование реализуется сравнением верхушки Mag с правыми частями правил, начиная с самого

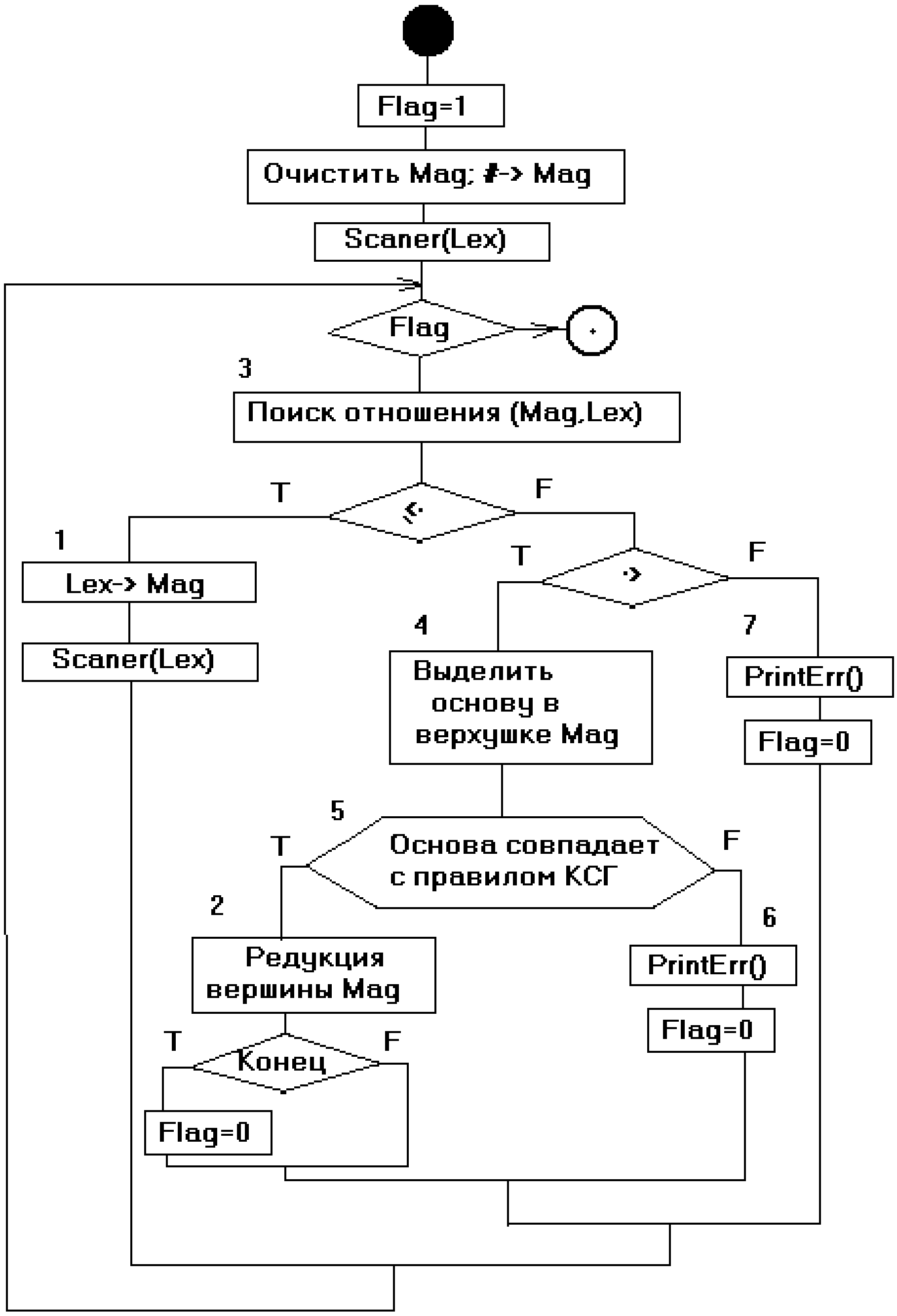


Рис. 8.9: Структурная схема анализатора предшествования

длинного. Причем при слабом предшествования блоки (4) и (5) сливаются в общую цепочку сравнений.

В зависимости от ситуации регистрируются различные ошибки:

а) в блоке (6) - "Неверная конструкция";

б) в блоке (7) - "Неверный символ lex".

Рассмотрим теперь способы уменьшения времени работы программы. Поиск отношений можно реализовать с помощью оператора *switch* языка Си по верхушке Mag с оператором *switch* по lex внутри. В результате получится работоспособная программа, требующая, однако, достаточно много сравнений. Можно сократить их число. Заметим, что после записи в магазин известна верхушка Mag, а любая правильная ветвь цикла заканчивается записью в магазин лексемы (при отношении "$\leq$") или нетерминала (при редукции). Следовательно, после записи известна та строка матрицы, которая будет анализироваться на следующем шаге. Если каждой строке матрицы поставить в соответствие собственный участок программы, то после записи элемента в магазин можно выполнить оператор *goto* на соответствующий фрагмент программы. Программа представляет собой совокупность помеченных участков, каждый из которых реализует одну строку матрицы и имеет вид:

```
<метка строки> :
switch (lex)
{
case < лексема для отношения "<" или  "=" >:
              <запись lex в магазин>;
              t=Scaner(lex);
              goto <метка для Mag>;
              //в магазине лексема, которую мы только что записали
case < лексема для отношения ">" >:
             <редукция в верхушке магазина>;
             goto <метка для Mag>;
             //в магазине новый нетерминал, который получили при редукции
default:     PrintError();
}
```

8.8 Контрольные вопросы к разделу 6

1. Какую стратегию разбора реализуют анализаторы предшествования?

2. Дайте определение отношений предшествования $<\cdot$, $\doteq$, $\cdot>$.

3. Чем похожи отношения предшествования $<\cdot$ и $\doteq$? В чем заключается различие между этими отношениями?

4. Постройте матрицу предшествования для КС–грамматики

$$G: \begin{array}{l} S \to bSa|A \\ A \to aAb|ab. \end{array}$$

5. Между какими символами появится конфликт в КС–грамматике

$$G: \begin{array}{l} S \to abSba|A \\ A \to aAb|b. \end{array}$$

Почему появляется конфликт?

6. В чем разница между простым и операторным предшествованием?

7. Зачем используются функции предшествования?

8. Дайте определение функций предшествования.

9. В чем преимущества использования функций слабого предшествования?

10. Для какого вида предшествования получится самая короткая программа? Дайте обоснование своего ответа.

11. Как реализовать в программе синтаксического анализа, построенной для метода слабого предшествования, выбор правила, по которому выполняется редукция?

12. Что нужно сделать в программе синтаксического анализатора, если обнаружено отношение $< \cdot$?

13. Имеется КС–грамматика, для которой решено написать анализатор, работающий по принципу операторного предшествования. Всегда ли можно выполнять редукцию по самому длинному правилу? Какие ограничения на КС–грамматику должны выполняться?

14. Как в программе синтаксического анализатора реализуется понятие "прозрачный нетерминал"? Какие способы его реализации существуют?

15. Почему при реализации операторного предшествования смежные нетерминалы могут вызвать сложности?

16. Зачем используется расширенное предшествование?

17. Что Вы можете сказать о сложности программы, которая реализует расширенное предшествование? Если такая программа является очень сложной, зачем все же вводят в рассмотрение расширенное предшествование?

18. Что называется графом линеаризации? Как используется граф линеаризации при построении функций предшествования?

19. Если программа синтаксического анализатора строится неявным методом, помогут ли при ее разработке функции предшествования?

20. При каком методе программирования синтаксического анализатора эффективно использование функций слабого предшествования? В чем преимущество функций слабого предшествования перед функциями предшествования?

8.9 Тесты для самоконтроля к разделу 6

1. Дайте определение отношения $A \cdot > B$.

Варианты ответов:

а) $A \cdot > B$, где $A \in V_T \cup V_N$, $B \in V_T$, если для некоторого нетерминала T имеется правило $T \to z_1 K D z_2$, существуют выводы $K \overset{+}{\Rightarrow} xA$ и $D \overset{*}{\Rightarrow} By$

б) $A \cdot > B$, где $A, B \in V_T \cup V_N$, если для некоторого нетерминала T имеется правило $T \to z_1 K D z_2$, существуют выводы $K \overset{+}{\Rightarrow} xA$ и $D \overset{*}{\Rightarrow} By$

в) $A \cdot > B$, где $A \in V_T$, $B \in V_T$, если существует вывод из аксиомы $S \overset{*}{\Rightarrow} z_1 A B z_2$.

г) $A \cdot > B$, где $A, B \in V_T \cup V_N$, если $S \overset{*}{\Rightarrow} xAD$, $D \overset{+}{\Rightarrow} By$

д) $A \cdot > B$, где $A \in V_N$, $B \in V_T$, если $S \overset{*}{\Rightarrow} xAD$, $D \overset{+}{\Rightarrow} By$

Правильный ответ: а.

2. Чем простое предшествование отличается от слабого предшествования?

Варианты ответов:

а) Простое предшествование требует различия всех правых частей правил грамматики, а для слабого предшествования эти ограничения снимаются.

б) Слабое предшествование построено на основе слияния двух знаков простого отношения $< \cdot$ и $\doteq$

в) Слабое предшествование построено на основе слияния двух знаков простого отношения $\cdot >$ и $\doteq$

г) Простое предшествование основано на построении полного дерева разбора, а слабое — скелета дерева разбора.

д) Для простого предшествования отношения стоятся между всеми символами, а для слабого — только между терминальными.

Правильный ответ: б.

3. Как устранить конфликт $a\cdot > b$ и $a \doteq b$?

Варианты ответов:

а) единственный известный метод устранения такого конфликта основан на использовании метода двойников для каждого символа всех правил грамматики;

б) единственный известный метод устранения такого конфликта основан на использовании метода двойников для каждого символа тех правил грамматики, которые являются причиной конфликта;

в) каждое вхождение символов a и b в правые части правил грамматики заменяется на двойник;

г) правило $A \to xaby$ заменяется на два эквивалентных $A \to Tby$ и $T \to xa$,

д) для правила $A \to xaby$ выполняется замена на два эквивалентных правила $A \to xaT$ и $T \to by$;

е) если существует правило $A \to xaby$, то для этого правила выполняется замена на два эквивалентных правила $A \to xaT$ и $T \to by$; в противном случае используется метод двойников.

Правильный ответ: д.

4. Зачем используются функции предшествования?

Варианты ответов:

а) для устранения некоторых конфликтов в матрице предшествования;

б) для сокращения объемов программы при программировании анализатора неявным способом;

в) для сокращения объемов используемой памяти при программировании анализатора явным способом;

г) для повышения быстродействия анализатора, построенного явным способом;

д) для повышения быстродействия анализатора, построенного неявным способом;

е) для повышения качества обнаружения ошибок.

Правильный ответ: в.

5. Чем отличаются функции простого предшествования от функций слабого предшествования?

Варианты ответов:

а) функции слабого предшествования позволяют повысить быстродействие анализатора по сравнению с анализатором, который основан на функциях простого предшествования;

б) функции слабого предшествования сокращают объем программы при программировании анализатора неявным способом;

в) функции простого предшествования нельзя построить, если в графе линеаризации имеются циклы, а при переходе с слабому предшествованию такие циклы всегда устраняются;

г) функции слабого предшествования занимают в памяти меньше места по сравнению с функциями простого предшествования;

е) функции слабого предшествования позволяют повысить качество обнаружения ошибок.

Правильный ответ: е.

8.10 Упражнения к разделу

8.10.1 Задание

Цель данного задания — написать программу восходящего анализатора одним из методов предшествования. Чтобы выполнить задание, сначала нужно выбрать метод анализа, приемлемый для заданной КС–грамматики. Затем Вы должны построить управляющую таблицу, при наличии конфликтов выбрать метод их устранения. И только тогда можно программировать синтаксический анализатор. Работу над заданием можно выполнить последовательно по следующим этапам.

1. Проанализировать методы восходящего анализа с точки зрения

— простоты реализации;

— требований к правилам КС-грамматики.

Выбрать метод восходящего анализа, подходящий для КС–грамматики языка программирования Вашего задания.

2. Если КС-грамматика содержит правила, конфликтующие с выбранным методом восходящего анализа, выполнить преобразование КС–грамматики.

3. Построить управляющую таблицу по Вашей КС–грамматике.

4. Проанализировать построенную управляющую таблицу и выделить все конфликты в этой таблице.

5. Для каждого конфликта в управляющей таблице определить методы его устранения:

— преобразование КС–грамматики;

— расширение контекста в программе анализатора;

— внесение ограничений в язык программирования, компилятор с которого Вы разрабатываете.

6. Для каждого конфликта реализовать выбранный метод его устранения.

7. Результатом Вашей работы должна стать бесконфликтная таблица, готовая для ее программирования: каждая клетка либо содержит не более одного отношения или действия, либо в клетке несколько элементов, но однозначно определены условия выбора одного из них (при этом допускается анализ ограниченного контекста или ограниченной цепочки в верхушке магазина).

8. Определить тип данных для представления одного элемента магазина синтаксического анализатора. Описать магазин в программе анализатора.

9. Написать основной цикл алгоритма восходящего анализатора.

10. Для каждого символа Вашей КС–грамматики, анализ которого предусмотрен алгоритмом, определить участок программы, соответствующий этому символу.

11. Построить фрагменты программы для обработки каждого символа в соответствии с управляющей таблицей.

12. Встроить в программу разрешение конфликтов в соответствии с предложенными Вами методами.

13. Построить проект, включающий стандартные определения, сканер, синтаксический анализатор, главную программу транслятора.

14. Отладить программу на правильных и ошибочных текстах программ.

Глава 9

$LR(k)$–ГРАММАТИКИ И $LR(k)$–АНАЛИЗАТОРЫ

9.1 Определение LR(k)-грамматики

Рассмотрим теперь общий случай восходящего анализа и выясним условия, при которых не более k очередных отсканированных символов позволят однозначно выполнить редукцию. КС–грамматика $G = (V_T, V_N, P, S)$ называется $LR(k)$–грамматикой, если для произвольного правого вывода

$$S = x_0 \Rightarrow x_1 \Rightarrow x_2 \Rightarrow ... \Rightarrow x_m = y$$

в каждой правовыводимой сентенциальной форме x_i можно выделить основу и определить нетерминал, на который она должна быть заменена, зная не более k символов справа от основы. Дадим формальное определение $LR(k)$–грамматики. Сразу сделаем одно существенное замечание: будем рассматривать разбор только в пополненной грамматике.

Определение 9.1. Для КС–грамматики $G = (V_T, V_N, P, S)$ пополненной грамматикой называется грамматика

$$G_1 = (V_T, V_N \cup \{F | F \notin V_N\}, P \cup \{F \to S\}, F).$$

Очевидно, что в соответствии с определением для пополнения заданной грамматики необходимо ввести в множество нетерминалов грамматики G новый нетерминал F, для которого имеется единственное правило $F \to S$. Новый нетерминал F становится аксиомой полученной грамматики. Цель пополнения грамматики заключается в том, чтобы при восходящем разборе без дополнительных проверок установить факт завершения грамматического разбора: если в магазине аксиома грамматики, значит разбор завершен.

Определение 9.2. КС–грамматика $G = (V_T, V_N, P, S)$ называется $LR(k)$–грамматикой для некоторого фиксированного целого числа k, если из условий существования правых выводов

$$S \overset{*}{\Rightarrow} xAy \Rightarrow \alpha\beta y,$$
$$S \overset{*}{\Rightarrow} uBz \Rightarrow \alpha\beta z,$$

и равенства $first_k(y) = first_k(z)$ следует, что $xAy = uBz$.

На первый взгляд приведенное определение устанавливает слишком жесткие ограничения на вывод: равенство $first_k(y) = first_k(z)$ требует полного совпадения сентенциальных форм. На самом деле, оно означает, что *предыстория разбора и очередные терминальные символы $first_k(y)$ однозначно определяют момент редукции.*

Разбор в $LR(k)$–грамматиках выполняется на основе управляющей таблицы, которая состоит из двух частей: таблицы переходов и таблицы действий. Строки таблицы — состояния анализатора. В таблице действий столбцы соответствуют очередным отсканированным цепочкам длины k, так, что на пересечении текущего состояния анализатора и текущей отсканированной цепочки указывается одно из выполняемых действий:

а) свертка по некоторому правилу грамматики,

б) перенос отсканированного символа в магазин,

в) окончание разбора.

В таблице переходов столбцы соответствуют элементам в верхушке магазина и определяют новое состояние анализатора после выполнения действия из таблицы действий. Таким образом, столбцы таблицы действий помечены цепочками длины k над множеством $V_T \cup \{\natural\}$, а столбцы таблицы переходов — символами из множества $V_T \cup V_N$.

Магазин $LR(k)$–анализатора предназначен для записи двух чередующихся элементов — состояний анализатора A_0, A_1,..., A_m и символов множества $V_T \cup V_N$. В начальный момент в магазин заносится начальное состояние A_0 и сканируется первый символ входной цепочки. Затем в цикле на каждом шаге выполняется действие из таблицы действий, строка которой определяется верхушкой магазина, а столбец — очередной отсканированной цепочкой длины k.

Если очередное действие — "свертка", то верхушка магазина (с учетом чередования состояний и символов) заменяется на правую часть правила, определяется новый переход по новому нетерминалу и состоянию в верхушке магазина. С учетом чередования в магазине состояний и символов при редукции по правилу $A \to x$ из магазина стирается $2|x|$ элементов.

Если очередное действие — "перенос", то отсканированная лексема заносится в магазин; из таблицы переходов в той же строке и столбце, определяемом текущей лексемой lex, в магазин переносится новое состояние анализатора, сканируется новый символ. Поскольку lex и верхушка магазина после записи туда лексемы совпадают, то можно считать, что переход и в этом случае определяется двумя верхними элементами магазина.

Структурная схема алгоритма $LR(1)$–анализатора приведена на рис. 9.1.

Пример 9.1. Для пополненной грамматики

$$\begin{aligned} G:\ \ & A \to S \\ & S \to SaSb|\varepsilon \end{aligned}$$

управляющая таблица $LR(1)$–анализатора имеет вид, представленный на рис. 9.2, а пример разбора цепочки $abab$ — на рис. 9.3.

9.2 Управляющая таблица LR(k)–анализатора

Будем предполагать, что знак "." отмечает место, в котором синтаксический анализатор смотрит на текущую сентенциальную форму.

Определение 9.3. $LR(k)$–ситуацией называется конструкция $[A \to w.u, x]$, где $A \to wu$ — правило грамматики, x — некоторая терминальная цепочка длины k,

Смысл определения $LR(k)$–ситуации заключается в том, что формализуется такое состояние восходящего анализатора, который уже прочитал справа цепочку длины k,

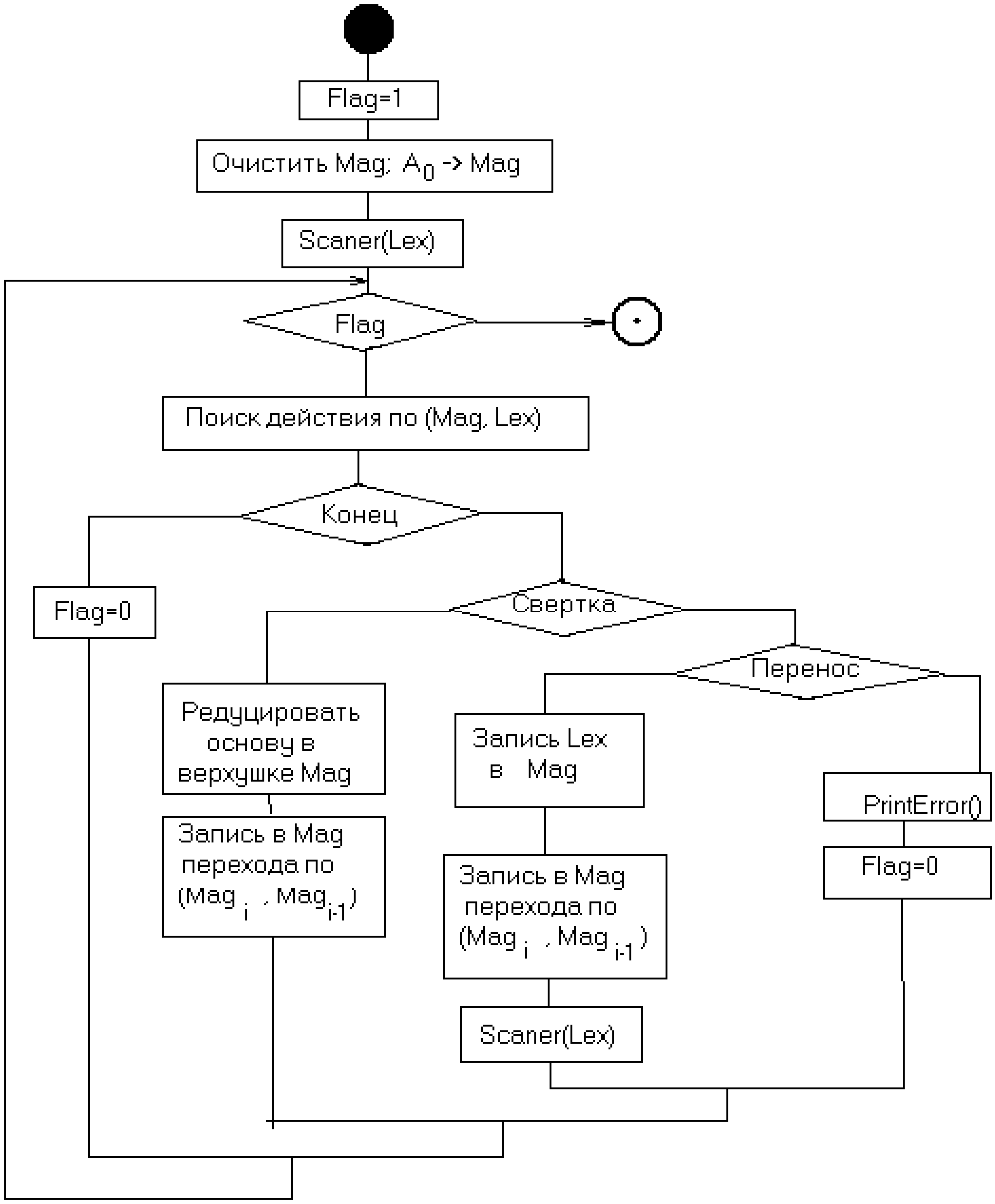


Рис. 9.1: Структурная схема $LR(1)$–анализатора

действия						переходы
состояния	a	b	$\natural$	S	a	b
A_0	свертка 2		свертка 2	A_1		
A_1	перенос		конец		A_2	
A_2	свертка 2	свертка 2		A_3		
A_3	перенос	перенос			A_4	A_5
A_4	свертка 2	свертка 2		A_6		
A_5	свертка 1		свертка 1			
A_6	перенос	перенос			A_4	A_7
A_7	свертка 1	свертка 1				

Рис. 9.2: Управляющая таблица LR(1)–анализатора

Вход	a		b		a		b		$\natural$	
					A_5				A_5	
					b				b	
				A_3	A_3			A_3	A_3	
				S	S			S	S	
			A_2	A_2	A_2		A_2	A_2	A_2	
			a	a	a		a	a	a	
		A_1	A_1	A_1	A_1	A_1	A_1	A_1	A_1	A_1
		S	S	S	S	S	S	S	S	S
	A_0	A_0	A_0	A_0	A_0	A_0	A_0	A_0	A_0	A_0

Рис. 9.3: Пример разбора цепочки по таблице LR(1)–анализатора

находится где–то в середине правила $A \to wu$, и в дальнейшем собирается выполнить редукцию по этому правилу (см. рис. 9.4).

Понятие $LR(k)$–ситуации в общем виде не учитывает уже проанализированного транслятором контекста. Поэтому для того, чтобы не накладывать чересчур жестких требований на грамматику, введем определение допустимой $LR(k)$–ситуации.

Определение 9.4. $LR(k)$–ситуация $[A \to w.u, x]$ называется допустимой для префикса yw, если

1) существует вывод $S\natural^k \overset{*}{\Rightarrow} yAz \Rightarrow ywuz$;

2)$x \in first_k(z)$.

Рассматривая определение 9.4 в свете процесса грамматического разбора можно сделать вывод о том, что цепочка y уже проанализирована и результат ее анализа находится в магазине, цепочка w подлежит анализу, а терминальная цепочка x представляет собой очередные отсканированные символы. Очевидно, что основной интерес представляют $LR(k)$–ситуации вида $[A \to w., x]$, т.к. они соответствуют редукции. С учетом того, что разбор выполняется в пополненной грамматике с аксиомой S, завершение процесса разбора определяется $LR(k)$–ситуацией $[S \to w., \natural^k]$.

Обозначим $V_k(x)$ — множество $LR(k)$–ситуаций, допустимых для x. Рассмотрим алгоритм построения всех $V_k(x)$. В начале разбора префиксом является пустая цепочка ε, следовательно, в начальный момент состояние может быть представлено

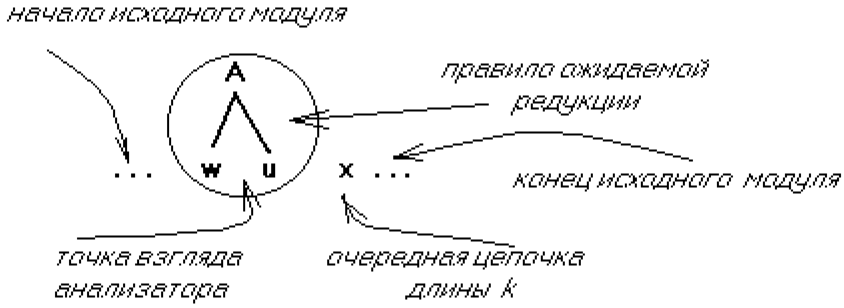


Рис. 9.4: Понятие LR(k)–ситуации $A \to w.u, x$

всевозможным набором $LR(k)$–ситуаций для пустого префикса. Построим сначала $V_0(\varepsilon)$:

1) для всех правил $S \to w$ включим в $V_k(\varepsilon)$ начальную $LR(k)$–ситуацию $[S \to .w, \natural]$, соответствующую началу процесса вывода;

2) если $[A \to .Bv, u] \in V_k(\varepsilon)$ и имеется правило грамматики $B \to z$, то для всех $x \in first_k(vu)$необходимо включить ситуацию $[B \to .z, x]$ в $V_k(\varepsilon)$.

Перейдем теперь к построению $V_k(a_1a_2...a_i)$ по $V_k(a_1a_2...a_{i-1})$. Очевидно, что такое построение должно позволить как перейти через символ a_i, так и применить правило к нетерминалу, перед которым находится "точка взгляда" синтаксического анализатора:

1) если $[A \to w.a_iv, u] \in V_k(a_1a_2...a_{i-1})$, то включим в $V_k(a_1a_2...a_i)$ новую $LR(k)$–ситуацию $[A \to wa_i.v, u]$, выполняя тем самым переход через нужный нам символ a_i;

2) если $[A \to w.Bv, u] \in V_k(a_1a_2...a_i)$ и имеется правило грамматики $B \to z$, то включим в $V_k(a_1a_2...a_i)$ новую $LR(k)$–ситуацию $[B \to .z, x]$ для всех $x \in first_k(vu)$, применяя тем самым правило $B \to z$.

Рассмотренное построение $V_k(a_1a_2...a_i)$ по $V_k(a_1a_2...a_{i-1})$ называется выполнением операции *goto*:

$$V_k(a_1a_2...a_i) = goto(V_k(a_1a_2...a_{i-1}), a_i).$$

Пример 9.2. Для грамматики

$$G: \begin{array}{l} A \to S \\ S \to SaSb|\varepsilon \end{array}$$

построим $V_1(\varepsilon)$. Для аксиомы включаем в $V_1(\varepsilon)$ начальную $LR(k)$–ситуацию

$$[A \to .S, \natural],$$

а затем в соответствии с правилами построения включим

$$[S \to .SaSb, \natural],\ [A \to .S, \natural],\ [S \to .SaSb, a],\ [S \to ., a].$$

Объединяя элементы, получим

$$V_1(\varepsilon) = [A \to .S, \natural],\ [S \to ., a|\natural],\ [S \to .SaSb, a|\natural].$$

Построим теперь с помощью операции *goto* остальные множества. Анализ $V_1(\varepsilon)$ показывает, что точка стоит перед символом S, поэтому

$$V_1(S) = goto(V_1(\varepsilon), S) = [A \to S., \natural],\ [S \to S.aSb, a|\natural].$$

Множества $V_1(a) = goto(V_1(\varepsilon), a)$ и $V_1(b) = goto(V_1(\varepsilon), b)$ являются пустыми, т.к. в $V_1(\varepsilon)$ отсутствуют элементы со знаком "." перед символами a и b.

Как уже отмечалось, состояние $LR(k)$–анализатора в начальный момент определяется множеством допустимых $LR(k)$–ситуаций для пустого префикса. Обозначим $A_0 = V_k(\varepsilon)$. Начиная с множества A_0 для любых символов из $V_T \cup V_N \cup \{\natural\}$ выполняя операцию *goto* будем получать новые множества допустимых ситуаций. Каждое такое множество A_i описывает текущее состояние $LR(k)$–анализатора и, следовательно, определяет соответствующие переходы в таблице переходов.

Пример 9.3. Для грамматики

$$\begin{array}{l} G : A \to S \\ S \to SaSb|\varepsilon \end{array}$$

построим множества A_i. Множество $V_1(\varepsilon)$ мы уже построили в примере 9.2. Построим теперь все остальные множества:

$$\begin{array}{lll}
A_0 = & \{[A \to .S, \natural], & \\
 & [S \to ., a|\natural], & \\
 & [S \to .SaSb, a|\natural]\}, & \\
A_1 = & goto(A_0, S) = & \{[A \to S., \natural], \\
 & & [S \to S.aSb, a|\natural]\}, \\
 & goto(A_0, a) = \emptyset, & \\
 & goto(A_0, b) = \emptyset, & \\
 & goto(A_1, S) = \emptyset, & \\
A_2 = & goto(A_1, a) = & \{[S \to Sa.Sb, \natural|a], \\
 & & [S \to .SaSb, a|b], \\
 & & [S \to ., a|b]\}, \\
A_3 = & goto(A_2, S) = & \{[S \to SaS.b, \natural|a], \\
 & & [S \to S.aSb, b|a]\}, \\
A_4 = & goto(A_3, a) = & \{[S \to Sa.Sb, b|a], \\
 & & [S \to .SaSb, a|b], \\
 & & [S \to ., a|b]\}, \\
A_5 = & goto(A_3, b) = & \{[S \to SaSb., \natural|a]\}, \\
A_6 = & goto(A_4, S) = & \{[S \to SaS.b, b|a], \\
 & & [S \to S.aSb, b|a]\}, \\
 & goto(A_6, a) = & \{[S \to Sa.Sb, b|a], \\
 & & [S \to .SaSb, a|b], \\
 & & [S \to ., a|b]\} \\
 & & \text{— совпадает с уже построенным множеством } A_4, \\
A_7 = & goto(A_6, b) = & \{[SaSb., b|a]\}.
\end{array}$$

Полученные множества позволяют сформировать таблицу переходов, приведенную в примере 9.1.

Перейдем теперь к алгоритму построения таблицы действий $LR(k)$–анализатора. Строки таблицы — состояния, соответствующие построенным множествам A_i, а столбцы соответствуют очередным отсканированным цепочкам длины k. На пересечении текущего состояния анализатора и текущей отсканированной цепочки указывается выполняемое действие: свертка по некоторому правилу грамматики, перенос отсканированного символа в магазин или окончание разбора. Последовательно будем формировать строки для состояний A_i. Для простоты рассмотрим $k = 1$.

Введем сначала в рассмотрение функцию $eff_k(y)$, являющуюся аналогом функции $first_k(y)$, но отличающуюся от нее тем, что в множество терминальных начал цепочек, выводимых из y, не включаются те, которые получены с использованием правила с пустой правой частью для самого левого нетерминала при выводе из y. Указанное отличие функции eff_k от функции $first_k$ соответсвует и названию данной функции: eff — это сокращение от $\varepsilon - free - first$ (ε — свободные начала цепочек).

Очевидно, что элемент строки для цепочки $\natural$ соответствует действию "конец", если $[S \to x., \natural] \in A_i$. Элемент строки для цепочки a определяет действие "свертка по правилу j", если $[B \to x., a] \in A_i$ и $B \to x$ является j–ым правилом грамматики. Элемент строки для цепочки a определяет действие "перенос", если $[B \to x.y, u] \in A_i$, а цепочка y — не пустая цепочка (в случае $y = \varepsilon$ выполняется свертка, а не перенос) и $a \in eff_k(yu)$.

Например, строка A_0 для грамматики из примера 9.3 заполняется действием "свертка 2" для столбцов $\natural$ и a , т.к. $[S \to ., \natural|a] \in A_0$. Для строки A_1 заносим действие "допуск" для символа $\natural$, т.к. $[A \to S., \natural] \in A_1$, и действие "перенос" для символа a, т.к. $[S \to S.aSb, \natural|a] \in A_1$.

9.3 Контрольные вопросы к разделу 7

1. В чем различие между $LL(k)$ и $LR(k)$ анализаторами?

2. Какую стратегию разбора использует $LR(k)$–анализатор?

3. Зачем при построении $LR(k)$–анализатора используется пополненная грамматика?

4. Что называется $LR(k)$ ситуацией?

5. Какую структуру имеет управляющая таблица $LR(k)$–анализатора?

6. Что хранится в магазине $LR(k)$–анализатора?

7. В чем смысл операции $goto(A_i, B)$? Как определяется эта операция? Какие параметры у этой операции?

8. По какому алгоритму строится множество $LR(k)$ ситуаций?

9. Как по множеству $LR(k)$ ситуаций строится таблица действий управляющей таблицы?

10. Как по множеству $LR(k)$ ситуаций строится таблица переходов управляющей таблицы?

11. Пусть в множестве $LR(k)$ ситуаций $V_k(x)$ имеется $LR(k)$–ситуация $[A \to w.a_iv, u]$, то что Вы можете сказать о множестве $V_k(xa_i)$?

12. Как достраивается множество $LR(k)$ ситуаций $V_k(x)$ в соотвествии с правилами грамматики?

13. Пусть $[A \to w.Bv, u] \in V_k(x)$ и имеется правило грамматики $B \to z$. Что Вы можете сказать о включении в $V_k(x)$ новых $LR(k)$–ситуаций?

14. Какие действия программируются в $LR(k)$–анализаторе, если в таблице действий стоит команда "перенос" ?

15. Какие действия программируются в $LR(k)$–анализаторе, если в таблице действий стоит команда "свертка" ?

16. Нарисуйте структурную схему $LR(k)$–анализатора.

17. Постройте управляющую таблицу $LR(k)$–анализатора для грамматики

$$\begin{aligned} G:\quad & S \to aSb|aS \\ & A \to Aab|\varepsilon \end{aligned}$$

18. Чем функция $eff_k(y)$, отличается от функции $first_k(y)$? Зачем вводится эта функция?

19. Как и чем определяется начальное состояние $LR(k)$–анализатора? Какой $LR(k)$–ситуации оно соответствует?

20. В какой момент $LR(k)$–анализатор обнаруживает ошибку в транслируемом исходном модуле?

9.4 Тесты для самоконтроля к разделу 7

1. Дайте определение $LR(k)$–грамматики.

Варианты ответов:

а) КС–грамматика $G = (V_T, V_N, P, S)$ называется $LR(k)$–грамматикой для некоторого фиксированного целого числа k, если из условий существования правых выводов

$$S \stackrel{*}{\Rightarrow} xAy \Rightarrow xwy,$$
$$S \stackrel{*}{\Rightarrow} uBz \Rightarrow xwz,$$

и равенства $first_k(wy) = first_k(wz)$ следует, что $x = u$.

б) КС–грамматика $G = (V_T, V_N, P, S)$ называется $LR(k)$–грамматикой для некоторого фиксированного целого числа k, если для любых двух ее правил $A \to x$ и $B \to x$ из существования двух левых выводов

$$S \stackrel{*}{\Rightarrow} wAz \Rightarrow wxz \stackrel{*}{\Rightarrow} wq,$$
$$S \stackrel{*}{\Rightarrow} wBz \Rightarrow wxz \stackrel{*}{\Rightarrow} wp,$$

для которых $first_k(q) = first_k(p)$, следует $A = B$.

в) КС–грамматика $G = (V_T, V_N, P, S)$ называется $LR(k)$–грамматикой для некоторого фиксированного целого числа k, если из условий существования правых выводов

$$S \stackrel{*}{\Rightarrow} xAy \Rightarrow xwy,$$
$$S \stackrel{*}{\Rightarrow} uBz \Rightarrow xwz,$$

и равенства $first_k(y) = first_k(z)$ следует, что $xAy = uBz$.

г) КС–грамматика $G = (V_T, V_N, P, S)$ называется $LR(k)$–грамматикой для некоторого фиксированного целого числа k, если для любых двух ее правил $A \to x$ и $A \to y$ из существования двух левых выводов

$$S \stackrel{*}{\Rightarrow} wAz \Rightarrow wxz \stackrel{*}{\Rightarrow} wq,$$
$$S \stackrel{*}{\Rightarrow} wAz \Rightarrow wyz \stackrel{*}{\Rightarrow} wp,$$

для которых $first_k(q) = first_k(p)$, следует $x = y$.

д) КС–грамматика $G = (V_T, V_N, P, S)$ называется $LR(k)$–грамматикой для некоторого фиксированного целого числа k, если из условий существования правых выводов

$$S \stackrel{*}{\Rightarrow} xAy \Rightarrow xwy,$$
$$S \stackrel{*}{\Rightarrow} xBz \Rightarrow xuz,$$

и равенства $first_k(y) = first_k(z)$ следует, что равенство правил $A \to w$ и $B \to u$.

Правильный ответ: в.

2. Дайте определение $LR(k)$–ситуции.

Варианты ответов:

а) $LR(k)$–ситуацией называется конструкция $[A \to w.u, x]$, где $A \to wu$ — правило грамматики, x — некоторая терминальная цепочка длины k,

б) $LR(k)$–ситуацией называется конструкция $[w.u, x]$, где $A \to wu$ — правило грамматики, x — некоторая цепочка длины k,

в) $LR(k)$–ситуацией называется конструкция $[A \to w.u, z]$, где $A \to wu$ — правило грамматики, z — терминальная цепочка длины k, и существует вывод $S\natural^k \overset{*}{\Rightarrow} yAz \Rightarrow ywuz$.

г) $LR(k)$–ситуацией называется конструкция $[A \to w.u, x]$, где $A \to wu$ — правило грамматики, x — терминальная цепочка длины k, и существует вывод $S\natural^k \overset{*}{\Rightarrow} yAz \Rightarrow ywuz$, причем $x \in first_k(z)$.

д) $LR(k)$–ситуацией называется конструкция $[A \to w.u, z]$, где $A \to wu$ — правило грамматики, z — терминальная цепочка длины k, и существует вывод $S \overset{*}{\Rightarrow} yAz \Rightarrow ywuz$.

Правильный ответ: а.

3. Как определяется клетка управляющей таблицы $LR(1)$–анализатора, в которой стоит действие "конец"?

Варианты ответов:

а) элемент строки управляющей таблицы для цепочки a и состояния A_i соответствует действию "конец", если $[S \to, x, a] \in A_i$.

б) элемент строки управляющей таблицы для аксиомы S и состояния A_i соответствует действию "конец", если $[S \to x., \natural] \in A_i$.

в) элемент строки управляющей таблицы для аксиомы S и состояния A_i соответствует действию "конец", если $[S \to, x, \natural] \in A_i$.

г) элемент строки управляющей таблицы для символа $\natural$ и состояния A_i соответствует действию "конец", если $[S \to x., \natural] \in A_i$, где S — аксиома пополненной грамматики.

д) элемент строки управляющей таблицы для аксиомы S пополненной грамматики и состояния A_i соответствует действию "конец", если $[S \to x., a] \in A_i$.

е) элемент строки управляющей таблицы для символа $\natural$ и символа $\natural$ соответствует действию "конец", если $[S \to x, , \natural] \in A_i$, где S — аксиома пополненной грамматики.

Правильный ответ: г.

4. Как определяется клетка управляющей таблицы, в которой стоит действие "свертка"?

Варианты ответов:

а) Элемент строки S для цепочки $\natural$ соответствует действию "свертка", если $[S \to x., \natural] \in A_i$.

б) Элемент строки A_i для цепочки a определяет действие "свертка", если $[B \to x.y, u] \in A_i$, а цепочка y — не пустая цепочка.

в) Элемент строки A_i для цепочки a определяет действие "свертка по правилу j", если $[B \to x.y, \natural] \in A_i$ и $B \to xy$ является j–ым правилом грамматики.

г) Элемент строки A_i для цепочки a определяет действие "свертка по правилу j", если $[B \to x., \natural] \in A_i$ и $B \to x$ является j–ым правилом грамматики.

д) Элемент строки B для цепочки a определяет действие "свертка по правилу j", если $[B \to x.y, a] \in A_i$ и $B \to xy$ является j–ым правилом грамматики.

е) Элемент строки A_i для цепочки a определяет действие "свертка по правилу j", если $[B \to x., a] \in A_i$ и $B \to x$ является j–ым правилом грамматики.

Правильный ответ: е.

5. Как определяется клетка управляющей таблицы, в которой стоит действие "перенос"?

Варианты ответов:

а) Элемент строки A_i для цепочки a определяет действие "перенос", если $[B \to x.y, u] \in A_i$.

б) Элемент строки A_i для цепочки a определяет действие "перенос", если $[B \to x.y, u] \in A_i$, а цепочка y — не пустая цепочка.

в) Элемент строки B для цепочки a определяет действие "перенос", если $[B \to x.y, u] \in A_i$, а цепочка y — не пустая цепочка.

г) Элемент строки A_i для цепочки $\natural$ соответствует действию "перенос", если $[S \to x., \natural] \in A_i$.

д) Элемент строки B для цепочки a и строки A_i определяет действие "перенос", если $[B \to x.y, u] \in A_i$, а цепочка $y = \varepsilon$.

Правильный ответ: б.

ЗАКЛЮЧЕНИЕ

Рассмотренные способы проектирования лексических и синтаксических анализаторов дают возможность не только разработать и строго обосновать правильность программы анализатора, но и являются основой для автоматизации самого процесса программирования анализаторов. Существует множество систем автоматизации проектирования языковых процессоров, основанных на формальных описаниях либо грамматик, либо других эквивалентных их представлениях. Современные технологии проектирования программного обеспечения, в частности проектирование языковых процессоров, обеспечивают высокую степень автоматизации труда программиста, что способствует повышению производительности труда и качества программных продуктов.

Дальнейшие алгоритмы и методы построения компиляторов будут рассмотрены во втророй части настоящего учебного пособия.

Основная литература

1. Ахо А., Лам М., Сети Р., Ульман Д.. Компиляторы: принципы, технологии и инструментарийю — М: "Вильямс", 2008, 768с.

2. Молчанов А. Системное программное обеспечение. — "Питер", 2010, 400с.

3. Свердлов С.З. Языки программирования и методы трансляции. — "Питер", 2007, 637 с.

4. Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. — М. "Вильямс", 2008, 378с.

Дополнительная литература

1. Ахо А., Ульман Дж.. Теория синтаксического анализа, перевода, компиляции. В 2 т. Т. 1,2. — М.: Мир, 1980.

2. Бек Л. Введение в системное программирование. - М.: Мир, 1988, 448 с.

3. Вирт Н. Алгоритмы и структуры данных. — СПб: "Невский диалект" , 2001, 351 с.

4. Гордеев А.В., Молчанова А.Ю. Системное программное обеспечение. — СПб, "Питер" , 2002, 736 с.

5. Керниган Б., Пайк Р. Практика программирования. — СПб: "Невский диалект", 2001, 380 с.

6. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. — М.: МЦНМО, 1999.

7. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. - М.: Мир, 1979, 654 с.

Оглавление

ВВЕДЕНИЕ **3**

1 ФОРМАЛЬНЫЕ ГРАММАТИКИ И ЯЗЫКИ **7**

1.1 **Понятие порождающей грамматики и языка** . . . 7
1.2 **Классификация грамматик** . . . 9
1.3 **Основные свойства КС–языков и КС–грамматик** . . . 10
1.4 **Грамматический разбор** . . . 12
1.5 **Преобразования КС–грамматик** . . . 18
1.5.1 **Правила с одним нетерминалом** . . . 18
1.5.2 **Правила с одинаковыми правыми частями** . . . 20
1.5.3 **Неукорачивающие грамматики** . . . 22
1.5.4 **Непродуктивные нетерминалы** . . . 24
1.5.5 **Независимые нетерминалы** . . . 25
1.5.6 **Терминальные правила** . . . 26
1.5.7 **Леворекурсивные и праворекурсивные правила** . . . 27
1.6 **Теорема о языке** $a^nb^nc^n$ . . . 28
1.7 **Контрольные вопросы к разделу** . . . 29
1.8 **Упражнения к разделу** . . . 30
1.8.1 **Задача** . . . 30
1.8.2 **Варианты заданий** . . . 32
1.9 **Тесты для самоконтроля к разделу** . . . 33

2 ЯЗЫКИ И АВТОМАТЫ **35**

2.1 **Понятие автомата и типы автоматов** . . . 35
2.2 **Формальное определение автомата** . . . 37
2.3 **Конечные автоматы** . . . 39
2.4 **Регулярные множества** . . . 40
2.5 **Минимизация конечных автоматов** . . . 42
2.6 **Операции над регулярными языками** . . . 44
2.7 **Автоматные грамматики и конечные автоматы** . . . 48
2.8 **Автоматы с магазинной памятью и КС–языки** . . . 53
2.9 **Разбор с возвратом** . . . 56
2.10 **Контрольные вопросы к разделу** . . . 60
2.11 **Упражнения к разделу** . . . 61
2.11.1 **Задача** . . . 61
2.11.2 **Варианты заданий** . . . 66
2.12 **Тесты для самоконтроля к разделу** . . . 67

3 ЛЕКСИКА, СИНТАКСИС И СЕМАНТИКА ЯЗЫКА **69**
3.1 Понятие языка программирования и языкового процессора . . 69
3.2 Структура компилятора . . . 73
3.3 Синтаксис языков программирования . . . 76
3.3.1 Программа . . . 77
3.3.2 Выражения . . . 78
3.3.3 Структурные операторы . . . 82
3.3.4 Составной оператор . . . 83
3.4 Контекстные условия языков программирования . . . 83
3.5 Типы синтаксических анализаторов . . . 87
3.6 Контрольные вопросы к разделу 1 . . . 87
3.7 Тесты для самоконтроля к разделу 1 . . . 88
3.8 Упражнения к разделу . . . 91
3.8.1 Задание . . . 91
3.8.2 Пример выполнения задания . . . 91
3.8.3 Варианты заданий . . . 95

4 ЛЕКСИЧЕСКИЙ АНАЛИЗ **99**
4.1 Формальные параметры функции сканера . . . 99
4.2 Таблица лексических единиц . . . 100
4.3 Программирование сканера . . . 103
4.3.1 Вариант 1 — простое состояние . . . 105
4.3.2 Вариант 2 — начальное состояние . . . 107
4.3.3 Вариант 3 — тупиковое заключительное состояние . . . 107
4.3.4 Вариант 4 — заключительное состояние с переходами . . 108
4.4 Простой пример программы сканера . . . 109
4.5 Отладка программы сканера . . . 111
4.6 Контрольные вопросы к разделу 2 . . . 116
4.7 Тесты для самоконтроля к разделу 2 . . . 116
4.8 Упражнения к разделу . . . 118
4.8.1 Задание . . . 118
4.8.2 Пример выполнения задания . . . 119

5 МЕТОД РЕКУРСИВНОГО СПУСКА **129**
5.1 Построение синтаксических диаграмм . . . 129
5.2 Преобразование синтаксических диаграмм . . . 133
5.2.1 Вынесение левых и правых множителей . . . 133
5.2.2 Удаление левой и правой рекурсии . . . 134
5.2.3 Подстановка диаграммы в диаграмму . . . 136
5.3 Разметка ветвей синтаксических диаграмм . . . 136
5.4 Алгоритм построения функций $first$, $last$ и $follow$ . . . 139
5.5 Программирование синтаксических диаграмм . . . 146
5.5.1 Простой нетерминальный или терминальный блок . . . 146
5.5.2 Ветвление . . . 147
5.5.3 Циклы . . . 148
5.5.4 Сообщения об ошибках . . . 151
5.6 Контрольные вопросы к разделу 3 . . . 152
5.7 Тесты для самоконтроля к разделу 3 . . . 153
5.8 Упражнения к разделу . . . 157
5.8.1 Задание . . . 157

5.8.2 Пример выполнения задания 158

6 КОНТЕКСТНЫЕ УСЛОВИЯ **172**
6.1 Структура таблиц компилятора 172
6.2 Информация в таблице компилятора 173
6.2.1 Простые переменные 174
6.2.2 Константы 175
6.2.3 Массивы 176
6.2.4 Структуры 177
6.2.5 Функции и процедуры 178
6.2.6 Метки 179
6.2.7 Типы 179
6.2.8 Программирование таблицы компилятора 180
6.3 Семантические подпрограммы и их вызовы 185
6.4 Трансляция описаний 188
6.4.1 Простые переменные и массивы 188
6.4.2 Функции 190
6.4.3 Структуры 191
6.5 Контрольные вопросы к разделу 4 192
6.6 Тесты для самоконтроля к разделу 4 192
6.7 Упражнения к разделу 194
6.7.1 Задание 194
6.7.2 Пример выполнения задания 195

7 $LL(k)$–ГРАММАТИКИ И $LL(k)$–АНАЛИЗАТОРЫ **204**
7.1 Определение LL(k) - грамматики 206
7.2 Управляющая таблица для LL(k) - грамматики 208
7.3 LL(k)-анализатор 212
7.4 Программа LL(k)-анализатора 212
7.5 S–анализаторы 216
7.6 Контрольные вопросы к разделу 5 219
7.7 Тесты для самоконтроля к разделу 5 219
7.8 Упражнения к разделу 221
7.8.1 Задание 221
7.8.2 Пример выполнения задания 222

8 АНАЛИЗАТОРЫ ПРЕДШЕСТВОВАНИЯ **239**
8.1 Простое предшествование 239
8.1.1 Понятие отношений предшествования 239
8.1.2 Формальное определение отношений предшествования . 240
8.1.3 Матрица предшествования 242
8.1.4 Алгоритм построения матрицы предшествования 243
8.2 Функции предшествования 245
8.3 Отношения и функции слабого предшествования 248
8.3.1 Определение отношений слабого предшествования 248
8.3.2 Построение функций слабого предшествования 249
8.4 Преобразование КС–грамматики к грамматике предшествования 250
8.5 Расширенное предшествование 252
8.6 Операторное предшествование 254

8.7 **Реализация анализаторов предшествования** 256
8.8 **Контрольные вопросы к разделу 6** 258
8.9 **Тесты для самоконтроля к разделу 6** 259
8.10 **Упражнения к разделу** 261
8.10.1 **Задание** 261

9 $LR(k)$–ГРАММАТИКИ И $LR(k)$–АНАЛИЗАТОРЫ **262**
9.1 **Определение LR(k)-грамматики** 262
9.2 **Управляющая таблица LR(k)–анализатора** 263
9.3 **Контрольные вопросы к разделу 7** 268
9.4 **Тесты для самоконтроля к разделу 7** 269

ЗАКЛЮЧЕНИЕ **272**