МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Чувашский государственный университет имени И.Н.Ульянова

Л.А.ПАВЛОВ

НИСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

Конспект лекций

Чебоксары 2003

УДК 681.3
П12

Павлов Л.А.
П12 Нисходящий синтаксический анализ: Конспект лекций/ Чуваш. ун-т. Чебоксары, 2003. 48 с.

Даны основы теории формальных языков и грамматик, классификация грамматик, приведены методы преобразования грамматик, позволяющие строить грамматики, ориентированные на нисходящий детерминированный синтаксический анализ, рассмотрены методы синтаксического анализа, основанные на рекурсивном спуске и построении специальных таблиц разбора.

Для студентов III курса факультета информатики и вычислительной техники специальности 220400 «Программное обеспечение вычислительной техники и автоматизированных систем», а также других инженерных специальностей для более глубокого изучения проблем информатики и вычислительной техники.

Ответственный редактор канд. техн. наук, доцент А.Л.Симаков

Утверждено Методическим советом университета

© Л.А.Павлов, 2003

1. ФОРМАЛЬНЫЕ ГРАММАТИКИ

Язык – это множество строк (предложений, цепочек), представляющих собой последовательность символов, каждый из которых принадлежит некоторому конечному *алфавиту* (*словарю языка*). Каждая строка языка формируется из словаря в соответствии с заданными правилами. Совокупность таких правил формирования называется *грамматикой* языка и определяет его структуру. Язык, используемый для описания грамматики какого-либо языка, называется *метаязыком*.

В описании языка различают синтаксис и семантику. *Синтаксис* – это множество формальных правил порождения правильно построенных строк языка. *Семантика* – это смысловое, содержательное значение каждой строки.

Для формального определения синтаксиса языков программирования (*формальных языков*) большое распространение получили такие метаязыки, как *синтаксические диаграммы* и *формы Бэкуса-Наура* (БНФ).

Строка есть конечная последовательность символов, каждый из которых принадлежит некоторому конечному алфавиту V; при этом символы в строке могут повторяться. Если строка содержит m символов, то говорят, что она имеет длину m. Строка длины 0, т.е. не содержащая ни одного символа, называется пустой и обозначается ε. Длина строки x обозначается $|x|$.

Пусть V – некоторый алфавит. Тогда через V^* (рефлексивно-транзитивное замыкание V) обозначается множество всех строк (включая пустую строку), составленных из символов, входящих в V, т. е. это множество определенных над алфавитом V строк. Через V^+ обозначается множество всех строк, исключая пустую строку, т. е. $V^* = V^+ \cup \{\varepsilon\}$.

Пример: $V = \{0, 1\}$.

$V^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, ...\}$.

$V^+ = \{0, 1, 00, 01, 10, 11, 000, ...\}$.

Если $x \in V^*$ – строка длины m, а $y \in V^*$ – строка длины n, то их объединение, обозначаемое xy, есть конкатенация (сцепление) строк x и y. В результате объединения получается строка длины $m + n$, т. е. $|xy| = m + n$. Например, если $x = a_1a_2...a_m$, а $y = b_1b_2...b_n$, тогда $xy = a_1a_2...a_mb_1b_2...b_n$. Объединение является ассоциативной операцией, т. е. $(xy)z = x(yz)$, но не является комму-

тативной операцией, т.к. в общем случае $xy \neq yx$. Для пустой строки, очевидно, справедливы следующие утверждения:

$\varepsilon x = x\varepsilon = x$ для любого $x \in V^*$.

Обычно совокупность строк, принадлежащих V^* и имеющих длину 2, обозначают V^2, имеющих длину 3 – соответственно V^3 и т. д.; V^0 – пустая строка. Тогда

$$V^+ = \bigcup_{i=1}^{\infty} V^i \text{ и } V^* = V^+ \cup \{\varepsilon\} = \bigcup_{i=0}^{\infty} V^i .$$

Формальным языком L над алфавитом V называется произвольное подмножество множества V^*. Если L_1, L_2 – два формальных языка, то их объединение $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$ также является формальным языком. Например, если $L_1 = \{11, 1\}$ и $L_2 = \{\varepsilon, a, bb\}$, то $L_1L_2 = \{11, 1, 11a, 1a, 11bb, 1bb\}$.

Объединение формального языка L с самим собой записывается как L^2, или, в общем случае, как

$$L^0 = \{\varepsilon\},\ L^1 = L,\ L^i = LL^{i-1} = L^{i-1}L \text{ для } i \geq 2.$$

Формально грамматика определяется как $G = (V_T, V_N, P, S)$, где V_T – конечное множество *терминальных* символов (*терминалов*), т.е. символов, принадлежащих собственно описываемому формальному языку; V_N – конечное множество *нетерминальных* символов (*нетерминалов*), т.е. символов, принадлежащих метаязыку (необходимо заметить, что у V_T и V_N нет общих символов, т.е. $V_T \cap V_N = \varnothing$); P – конечное множество *продукций* (*правил вывода*, *порождающих правил*) вида $\alpha \to \beta$, где α – левая часть продукции – это строка такая, что $\alpha \in (V_T \cup V_N)^+$, а β – правая часть – строка, такая, что $\beta \in (V_T \cup V_N)^*$; $S \in V_N$ – *начальный символ* (*аксиома*) грамматики.

Примем следующие соглашения об обозначениях. Терминалы будем представлять строчными буквами или, если они являются словами-символами языка, – выделять жирным шрифтом. Примеры терминалов: *a*, +, **begin**, **while**. Нетерминалы будем представлять прописными буквами или, если они являются многосимвольными словами, заключать их в угловые скобки. Примеры нетерминалов: *A*, <ОПЕРАТОР>.

Пусть дана грамматика $G = (\{a, b\}, \{S\}, P, S)$, где P представляет множество следующих продукций: $S \to aSb$, $S \to \varepsilon$, или, в более краткой форме записи: $S \to aSb \mid \varepsilon$.

Чтобы вывести предложение этого языка, поступают следующим образом. Начинают с начального символа S и заменяют его на aSb или ε. Если S опять появится в полученной строке, его опять можно заменить с помощью одного из этих правил, и т. д. Полученная таким образом любая строка, не содержащая S, является предложением этого языка. Последовательность таких шагов называется выводом строки (предложения) и обычно записывается следующим образом:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$.

Выводимая строка формально определяется следующим образом. Пусть $G = (V_T, V_N, P, S)$ – грамматика и пусть $\gamma_1\alpha\gamma_2 \in (V_T \cup V_N)^+$ – строка терминальных и нетерминальных символов длиной ≥ 1. Если $\alpha \to \beta$ – продукция из P, то подстрока α в строке может быть заменена строкой β, и в результате получится $\gamma_1\beta\gamma_2$. Это записывается следующим образом:

$$\gamma_1\alpha\gamma_2 \Rightarrow \gamma_1\beta\gamma_2,$$

при этом говорят, что строка $\gamma_1\alpha\gamma_2$ *генерирует* строку $\gamma_1\beta\gamma_2$, или что строка $\gamma_1\beta\gamma_2$ *выводится* из строки $\gamma_1\alpha\gamma_2$.

Если $\alpha_1, \alpha_2, ..., \alpha_n \in (V_T \cup V_N)^*$, и $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n$ ($n \geq 1$), то обычно пишут сокращенно $\alpha_1 \overset{+}{\Rightarrow} \alpha_n$, и при этом говорят, что строка α_n выводится из строки α_1 за один или более шагов. Аналогично $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$ означает, что строку α_n можно вывести из строки α_1 с помощью нуля или более применений правил грамматики.

Если строка $\alpha \in (V_T \cup V_N)^*$ такая, что $S \overset{*}{\Rightarrow} \alpha$, то строку α называют *сентенциальной формой* грамматики G. *Сентенцией* грамматики G называют произвольную сентенциальную форму из V_T^*, т.е. произвольную строку терминальных символов, которая может быть выведена из начального символа S. Тогда множество всех сентенций грамматики G называется языком, порожденным грамматикой G, и обозначается $L(G)$.

Таким образом, $L(G) = \{x \in V_T^* \mid S \overset{*}{\Rightarrow} x\}$. Если две различные грамматики G и G' порождают один и тот же язык, т.е. $L(G) = L(G')$, то грамматики G и G' *эквивалентны*.

Одной из стандартных классификаций грамматик является *иерархия Хомского*, основанная на наложении определенных ограничений на продукции грамматики.

Любая грамматика определенного выше вида называется *грамматикой типа 0* или *грамматикой общего вида*. На продукции не наложено никаких ограничений.

Если грамматика обладает тем свойством, что для всех продукций вида $\alpha \to \beta$ выполняется ограничение $|\alpha| \le |\beta|$, где $|\alpha|$ и $|\beta|$ – соответствующие длины строк, то такая грамматика называется *грамматикой типа 1* или *контекстно-зависимой*.

Если в грамматике все левые части продукций состоят из одного нетерминального символа, то ее называют *грамматикой типа 2*, или *контекстно-свободной*.

Если каждая продукция грамматики имеет одну из форм:

$A \to a$, $A \to aB$,

где a – терминальный символ, а A и B – нетерминальные, то ее называют *грамматикой типа 3, выровненной вправо*, или *регулярной*, или *автоматной*.

Выровненная влево грамматика определяется аналогично, т.е. все ее продукции имеют одну из форм

$A \to a$, $A \to Ba$,

она также является грамматикой типа 3.

Очевидно, что эта иерархия – включающая, т. е. регулярные грамматики являются контекстно-свободными, которые в свою очередь являются контекстно-зависимыми, и т. д.

Иерархии грамматик соответствует иерархия языков. Однако при этом необходимо учитывать следующий факт. Например, если язык генерируется посредством контекстно-зависимой грамматики, то это не обязательно означает, что язык только контекстно-зависимый и не может быть контекстно-свободным, поскольку, если язык является контекстно-свободным, то всегда можно определить эквивалентную контекстно-свободную грамматику.

Наибольшее практическое применение находят регулярные (на этапе лексического анализа) и контекстно-свободные (на этапе синтаксического анализа) грамматики, которые позволяют специфицировать большинство конструкций современных языков программирования и использовать в качестве распознавате-

лей строк языка достаточно простые средства. В частности, распознавателем регулярного языка является конечный автомат, а распознавателем контекстно-свободного языка – магазинный автомат (автомат с магазинной памятью).

2. КОНТЕКСТНО-СВОБОДНЫЕ ГРАММАТИКИ

Формальное определение основных синтаксических понятий большинства языков программирования может быть учтено с помощью БНФ или синтаксических диаграмм. При этом вид левой части каждой продукции может быть ограничен лишь единственным нетерминальным символом, т. е. синтаксис большинства языков программирования в своей основной части может быть определен с помощью контекстно-свободных грамматик.

Контекстно-свободной грамматикой (КС-грамматикой) называется грамматика $G = (V_T, V_N, P, S)$, каждая продукция которой имеет вид $A \to \beta$, где $A \in V_N$, $\beta \in (V_T \cup V_N)^*$. При соблюдении соглашений об обозначении терминалов и нетерминалов для задания грамматики достаточно определить множество продукций и указать начальный нетерминал. Если не оговорено особо, начальным нетерминалом будем считать нетерминал левой части первой продукции множества.

Любой язык, порожденный КС-грамматикой, называется контекстно-свободным языком. Термин "контекстно-свободный" обусловлен тем, что любой символ $A \in V_N$ в сентенциальной форме грамматики G может быть раскрыт согласно продукции $A \to \beta$ независимо от того, какими строками он окружен внутри самой сентенциальной формы.

В КС-грамматике любая строка $x \in L(G)$ может быть выведена из начального символа S. Общепринятым методом представления такого вывода является *дерево вывода* (*дерево грамматического разбора*, *синтаксическое дерево*).

Синтаксическое дерево строится по порождению $S \Rightarrow A_1A_2...A_n \Rightarrow ... \Rightarrow T_1T_2...T_k$, где $A_i \in (V_T \cup V_N)$, $i = 1, 2, \ldots, n$; $T_j \in V_T$, $j = 1, 2, \ldots, k$, следующим образом. Из корня дерева, помеченного начальным нетерминалом S, отходит n ветвей по числу символов в строке, непосредственно порожденной начальным нетерминалом. Каждая из n ветвей заканчивается вершиной, помеченной символом A_i. Вершины метятся слева направо в поряд-

ке возрастания номера индекса. Если в процессе порождения к нетерминалу A_i применена продукция $A_i \to B_1B_2...B_t$, то вершина A_i становится корнем поддерева, из которого выходит t ветвей, каждая из которых заканчивается вершиной, помеченной символом B_i, $i = 1, 2, \ldots, t$. Такое построение производится для всех вершин, помеченных нетерминальными символами. Построение дерева заканчивается, когда все листья оказываются помеченными терминалами $T_1, T_2, ..., T_k$. Совокупность этих меток при просмотре вершин слева направо образует терминальную строку (сентенцию) $T_1T_2...T_k$. Очевидно, что в полностью построенном синтаксическом дереве листья соответствуют терминалам, а внутренние вершины – нетерминалам.

Пример построения синтаксического дерева для грамматики G с продукциями

$S \to AB$,

$A \to aA|a$,

$B \to bB|b$

при выводе строки *aaabb* (в более краткой форме – a^3b^2) представлен на рис. 1. Это дерево соответствует последовательности выводов: $S \Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaaB \Rightarrow aaabB \Rightarrow aaabb$.

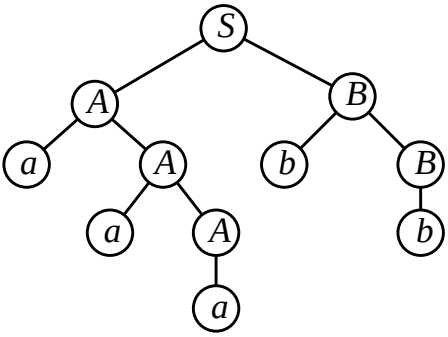

Рис. 1

Как правило, можно найти несколько возможных вариантов вывода одной и той же строки. Например, строку a^3b^2 можно вывести следующим образом:

$S \Rightarrow AB \Rightarrow AbB \Rightarrow aAbB \Rightarrow aaAbB \Rightarrow aaAbb \Rightarrow aaabb$,

$S \Rightarrow AB \Rightarrow aAB \Rightarrow aAbB \Rightarrow aAbb \Rightarrow aaAbb \Rightarrow aaabb$,

$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaAbB \Rightarrow aaabB \Rightarrow aaabb$.

Все эти схемы вывода соответствуют одному и тому же синтаксическому дереву.

Если для любой строки $x \in L(G)$ все возможные схемы его вывода соответствуют одному и тому же синтаксическому дереву, то такая КС-грамматика называется *однозначной*. Если же различным схемам вывода соответствуют несовпадающие деревья вывода, то грамматика называется *неоднозначной*.

Схема вывода строки $x \in L(G)$ называется *левосторонней*, если раскрывается всегда самый левый нетерминал сентенциальной формы. Поскольку теперь каждому синтаксическому дереву соответствует единственная левосторонняя схема вывода, то можно переопределить неоднозначную грамматику следующим образом: КС-грамматика G называется неоднозначной, если существует такая строка $x \in L(G)$, которой можно поставить в соответствие две различные левосторонние схемы вывода.

Аналогично можно определить *правостороннюю* схему вывода. Существуют также другие схемы вывода, не являющиеся ни лево-, ни правосторонними.

Для примера рассмотрим неоднозначную грамматику с продукциями $S \to SaS \mid b$.

Строку *babab* можно получить двумя различными левосторонними схемами вывода:

$S \Rightarrow SaS \Rightarrow SaSaS \Rightarrow baSaS \Rightarrow babaS \Rightarrow babab$,

$S \Rightarrow SaS \Rightarrow baS \Rightarrow baSaS \Rightarrow babaS \Rightarrow babab$.

Этим схемам вывода соответствуют различные синтаксические деревья, изображенные на рис. 2.

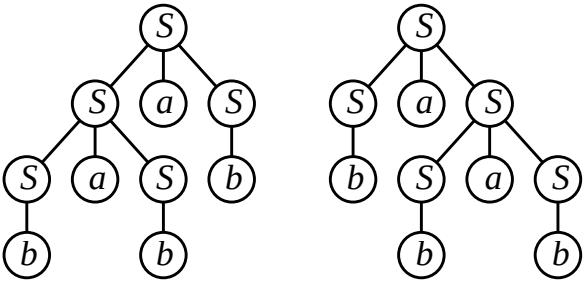

Рис. 2

Задача установления неоднозначности какой-либо грамматики является неразрешимой, т. е. не существует алгоритма, который принимал бы любую грамматику в качестве входа и определял бы, однозначна она или нет. В некоторые языки уже заложена неоднозначность. Это означает, что их нельзя генерировать с помощью однозначной грамматики. С другой стороны, некоторые неоднозначные грамматики можно преобразовать в однозначные, генерирующие тот же язык. Например, грамматика с продукциями $S \to Sab \mid b$ является однозначной и генерирует тот же язык, что и рассмотренная выше неоднозначная грамматика. Строка *bababab* получается следующей левосторонней схемой вывода: $S \Rightarrow Sab \Rightarrow Sabab \Rightarrow bababab$.

3. ПРЕОБРАЗОВАНИЕ ГРАММАТИК

При построении грамматик часто возникает необходимость в их эквивалентных преобразованиях для того, чтобы они удовлетворяли определенным критериям, но при этом не изменялся порождаемый язык. Рассмотрим некоторые простые, но наиболее часто применяемые и важные приемы преобразований контекстно-свободных грамматик.

3.1. Удаление бесполезных символов

Символ грамматики (терминал или нетерминал) называется *недостижимым*, если он не появляется ни в одной строке, выводимой из начального символа. Очевидно, что если нетерминал левой части продукции является достижимым, то и все символы правой части достижимы. На этом свойстве основана процедура выявления недостижимых символов, которую можно представить следующим образом:

1. Образовать одноэлементный список, состоящий из начального символа грамматики.

2. Если найдена продукция, левая часть которой уже имеется в списке, то включить в список все символы, содержащиеся в ее правой части.

Если на шаге 2 список не пополняется новыми символами, то получен список всех достижимых символов, а символы, не попавшие в список, являются недостижимыми.

Нетерминал называется *непроизводящим* (*бесплодным*), если он не порождает ни одной терминальной строки. Если же из не-

терминала можно вывести какую-нибудь терминальную строку, то он называется *продуктивным*. Очевидно, что если все символы правой части продукции продуктивные, то продуктивным является и нетерминал в левой части. На этом свойстве основана процедура выявления непроизводящих нетерминалов:

1. Составить список нетерминалов, для которых существует хотя бы одна продукция, правая часть которой не содержит нетерминалов.

2. Если найдена такая продукция, что все нетерминалы, стоящие в ее правой части, уже занесены в список, то добавить в список нетерминал из левой части.

Если на шаге 2 список больше не пополняется, то получен список всех продуктивных нетерминалов, а все не попавшие в него нетерминалы – непроизводящие.

В КС-грамматике с множеством продукций

$S \to aSb \mid dAc \mid a,$

$A \to cBe \mid dAf,$

$B \to aAa,$

$C \to ad$

непроизводящими являются нетерминалы A и B, а недостижимым – нетерминал C.

Символы, которые являются непроизводящими или недостижимыми, называются *бесполезными*. Исключение бесполезных символов из грамматики заключается в удалении сначала продукций, содержащих непроизводящие символы, а затем – продукций, содержащих недостижимые символы. Например, в рассмотренной выше грамматике сначала удаляются продукции

$S \to dAc,$

$A \to cBe \mid dAf,$

$B \to aAa,$

поскольку они содержат непроизводящие нетерминалы A и B, а затем – продукция $C \to ad$. В результате преобразований получается грамматика с множеством продукций $S \to aSb \mid a$.

Грамматика, не содержащая бесполезные символы, называется *приведенной*. В дальнейшем будем рассматривать только приведенные КС-грамматики.

3.2. Замена вхождений

Данное преобразование позволяет сократить число нетерминалов в грамматике и состоит в том, что если левая часть продукции входит в правую часть продукции R, то замена данного вхождения приведет просто к замене продукции R другой продукцией. Если такую замену выполнить для всех продукций с данным нетерминалом в левой части, то этот нетерминал можно исключить из грамматики. Исключением является случай, когда правая часть продукции содержит нетерминал левой части, например $S \to aSb|a$. В этом случае удаление нетерминала из грамматики невозможно.

Формально данное преобразование можно представить следующим образом. Если $A \to \alpha_1 B \alpha_2$ – продукция грамматики и $B \to \beta_1|\beta_2|\ldots|\beta_k$ – все продукции этой грамматики, содержащие в своих левых частях нетерминал B, тогда продукции вида $A \to \alpha_1 B \alpha_2$ можно поставить в соответствие продукции вида $A \to \alpha_1\beta_1\alpha_2|\alpha_1\beta_2\alpha_2|\ldots|\alpha_1\beta_k\alpha_2$.

Пусть дана грамматика с множеством продукций

$S \to AB|Bb|Ba$,

$A \to a$,

$B \to aSb|b$.

Выполним замену вхождений нетерминала B. В результате получим

$S \to AaSb|Ab|aSbb|bb|aSba|ba$,

$A \to a$.

Замена вхождения нетерминала A приведет к следующему множеству продукций:

$S \to aaSb|ab|aSbb|bb|aSba|ba$.

В общем случае замена вхождений приводит к увеличению числа продукций. Исключение представляет случай, когда заменяемый нетерминал является левой частью единственной продукции (в примере – нетерминал A).

Можно использовать и обратное преобразование, когда некоторая подстрока заменяется новым нетерминалом, с целью сокращения числа продукций.

3.3. Преобразование леворекурсивных продукций

Продукция вида $A \to A\alpha$, где $A \in V_N$, $\alpha \in (V_N \cup V_T)^*$, называется *леворекурсивной* (содержит *прямую левую рекурсию*), а продукция вида $A \to \alpha A$ – *праворекурсивной*. Для любой КС-грамматики, содержащей леворекурсивные продукции, можно построить эквивалентную КС-грамматику, не содержащую леворекурсивных продукций.

Преобразование заключается в следующем. Пусть грамматика содержит леворекурсивные продукции $A \to A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m$ и не являющиеся леворекурсивными продукции $A \to \beta_1 | \beta_2 | \ldots | \beta_n$, имеющие нетерминал A в своей левой части. Тогда новая эквивалентная грамматика может быть построена добавлением нового нетерминала A' и заменой леворекурсивных продукций продукциями вида

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

$A' \to \alpha_1 | \alpha_2 | \ldots | \alpha_m | \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A'$.

Таким образом, рассмотренное преобразование заменяет левую рекурсию на правую, и может быть выполнено для любой КС-грамматики.

Для иллюстрации техники преобразования рассмотрим грамматику (E – начальный символ):

$E \to E + T | T$,

$T \to T \times F | F$,

$F \to (E) | i$.

В соответствии с рассмотренным выше правилом продукции $E \to E + T | T$ преобразуются в продукции $E \to T | TE'$ и $E' \to + T | + TE'$, аналогично продукции $T \to T \times F | F$ преобразуются в продукции $T \to F | FT'$ и $T' \to \times F | \times FT'$. В результате получается грамматика без леворекурсивных продукций:

$E \to T | TE'$, $\quad T' \to \times F | \times FT'$,

$E' \to + T | + TE'$, $\quad F \to (E) | i$.

$T \to F | FT'$,

3.4. Исключение леворекурсивного цикла

Говорят, что грамматика имеет *леворекурсивный цикл*, если в грамматике имеется нетерминал A такой, что $A \overset{+}{\Rightarrow} A\alpha$, т. е. из нетерминала A можно вывести строку, начинающуюся с A. Заме-

тим, что понятие леворекурсивной продукции есть частный случай общего понятия леворекурсивного цикла.

Грамматику, содержащую леворекурсивный цикл, можно достаточно просто преобразовать в грамматику, содержащую только леворекурсивные продукции (прямую левую рекурсию), и далее исключить леворекурсивные продукции, преобразовав их в праворекурсивные. Чтобы заменить рекурсивный цикл на прямую рекурсию, необходимо:

1. Упорядочить нетерминалы грамматики, начиная с начального, т. е. $V_N = \{A_1, A_2, \ldots, A_m\}$, $m \geq 1$, где A_1 соответствует начальному нетерминалу. В результате продукции, у которых правая часть начинается с нетерминала, примут вид $A_i \to A_j\alpha$, где $\alpha \in (V_T \cup V_N)^*$. Если у всех продукций $i < j$, левая рекурсия в грамматике отсутствует, если $i = j$, данная продукция является леворекурсивной, если $i > j$, имеет место леворекурсивный цикл.

2. Для продукций вида $A_i \to A_j\alpha$, где $i > j$, производится замена вхождений нетерминала A_j. Такая последовательность замен повторяется до тех пор, пока не получится продукция, для которой $i = j$, т. е. пока рекурсивный цикл не сведется к прямой левой рекурсии.

В качестве примера такого преобразования рассмотрим следующую грамматику:

$S \to AS \mid AB,$

$A \to BS \mid a,$

$B \to SA \mid b.$

Прежде всего упорядочим нетерминалы грамматики. Для этого переименуем их следующим образом: S переименуем в A_1, A – в A_2 и B – в A_3. В результате продукции будут иметь вид:

$A_1 \to A_2A_1 \mid A_2A_3,$

$A_2 \to A_3A_1 \mid a,$

$A_3 \to A_1A_2 \mid b.$

Продукция $A_3 \to A_1A_2$ показывает, что в грамматике имеется леворекурсивный цикл. Произведем замену вхождений нетерминала A_1 следующим образом: $A_3 \to A_2A_1A_2 \mid A_2A_3A_2$. Поскольку условие $i = j$ еще не выполнено, продолжим процесс замены вхождений нетерминала A_2: $A_3 \to A_3A_1A_1A_2 \mid aA_1A_2 \mid A_3A_1A_3A_2 \mid aA_3A_2$. Получили леворекурсивные продукции, процесс замен прекращает-

ся. В результате получена грамматика, в которой вместо леворекурсивного цикла имеется прямая левая рекурсия

$S \to AS|AB$,

$A \to BS|a$,

$B \to BSSA|aSA|BSBA|aBA|b$.

3.5. Факторизация

Если в грамматике имеются продукции вида

$A \to \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_k$, $\alpha \in (V_N \cup V_T)^+$, $\beta_i \in (V_N \cup V_T)^*$, $1 \le i \le k$

с нетерминалом A в левой части, то эти продукции можно заменить, добавив новый нетерминал X, на следующие:

$A \to \alpha X$,

$X \to \beta_1|\beta_2|\ldots|\beta_k$.

Такое преобразование называется *факторизацией*.

Например, пусть дана грамматика с продукциями

$S \to aSb|aSc|d$.

Факторизации преобразует их в следующие продукции:

$S \to aSX|d$,

$X \to b|c$.

3.6. Удаление ε-продукций

Продукция вида $A \to \varepsilon$, называется ε-*продукцией* (*аннулирующей продукцией*). Грамматика называется ε-*свободной*, если она не имеет ε-продукций.

Следует заметить, что если пустая строка принадлежит языку, то избавиться от ε-продукции в ее грамматике, не изменяя порождаемого языка, невозможно. Самое большее, чего можно достигнуть, это преобразовать грамматику G таким образом, чтобы полученная грамматика G' была ε-свободной и выполнялось условие $L(G') = L(G) - \{\varepsilon\}$.

Для любой КС-грамматики $G = (V_T, V_N, P, S)$, содержащей ε-продукции, можно построить КС-грамматику $G' = (V_T, V_N, P', S)$, такую, что $L(G') = L(G) - \{\varepsilon\}$, следующим образом:

1. Объединить все имеющиеся в P продукции, за исключением ε-продукций, в множество P'.

2. Рассмотреть все нетерминалы $A \in V_N$, такие, что $A \overset{*}{\Rightarrow} \varepsilon$. Такие нетерминалы называются ε-*порождающими нетермина-*

лами. Каждой продукции $p \in P'$, у которой содержится в правой части ε-порождающий нетерминал, поставить в соответствие такую продукцию, что в ее правой части опущены (по сравнению с продукцией p) один или более нетерминалов, являющихся для грамматики G ε-порождающими нетерминалами, и присоединить ее к множеству P'. Другими словами, необходимо во все продукции грамматики выполнить все возможные подстановки пустой строки вместо ε-порождающего нетерминала.

Если пустая строка принадлежит языку, для получения эквивалентной грамматики необходимо добавить продукцию вида $S \to \varepsilon$, где S – начальный символ грамматики. В этом случае часто предъявляют дополнительное требование – символ S не должен встречаться в правых частях всех продукций грамматики. Чтобы выполнить это требование, можно использовать следующее простое преобразование. Если нетерминал S входит в правые части каких-либо продукций, то следует ввести новый начальный нетерминал S' и заменить продукцию $S \to \varepsilon$ на две новые продукции: $S' \to \varepsilon$ и $S' \to S$.

Проиллюстрируем преобразование на примере грамматики

$S \to ASB \mid \varepsilon,$

$A \to aA \mid \varepsilon,$

$B \to bB \mid b.$

Нетерминалы S и A являются ε-порождающими. Выполнение всех возможных замен этих нетеминалов пустой строкой приведет к следующим продукциям:

$S \to ASB \mid SB \mid AB \mid B,$

$A \to aA \mid a,$

$B \to bB \mid b.$

Поскольку пустая строка принадлежит языку, множество продукций следует дополнить продукцией $S \to \varepsilon$. Учитывая, что начальный нетерминал S входит в правые части некоторых продукций, заменим продукцию $S \to \varepsilon$ на продукции $S' \to \varepsilon$ и $S' \to S$. В результате получается следующая грамматика:

$S' \to \varepsilon \mid S,$

$S \to ASB \mid SB \mid AB \mid B,$

$A \to aA \mid a,$

$B \to bB \mid b.$

4. СИНТАКСИЧЕСКИЙ АНАЛИЗ

Процесс нахождения синтаксической структуры строки языка или, другими словами, процесс построения синтаксического дерева называется *синтаксическим анализом*. Синтаксическому анализатору предъявляется некоторая строка, и он должен определить, принадлежит ли она данному языку.

4.1. Магазинный автомат

Распознавателем контекстно-свободного языка является магазинный автомат (автомат с магазинной памятью), эквивалентный конечному автомату, к которому добавлен стек. Справедливо следующее утверждение: для любого КС-языка существует недетерминированный магазинный автомат, который принимает его, и наоборот – если некоторый недетерминированный магазинный автомат принимает некоторый язык, то этот язык является контекстно-свободным языком.

В функции магазинного автомата входит: а) чтение входного символа, замещение верхнего символа стека строкой символов (возможно пустой) и изменение состояния или б) все то же самое, но без чтения входного символа. Формально магазинный автомат определяется следующим образом:

$M = (K, T, V, \delta, k_0, z_0, F)$, где K – конечное *множество состояний*, T – конечный *входной алфавит*, V – конечный *стековый алфавит*, δ – функция переходов, $k_0 \in K$ – *начальное состояние* автомата, $z_0 \in V$ – *исходный символ стека*, который первоначально находится в стеке (обычно это маркер дна или нижней границы стека), $F \subseteq K$ – *множество конечных состояний*.

Функция переходов δ (называемая также магазинной функцией) отображает множество $K \times V \times (T \cup \{\varepsilon\})$ на множество конечных подмножеств множества $K \times V^*$.

Запись функции $\delta(k, A, a) = (k', Y)$, $k, k' \in K$, $A \in V$, $a \in T \cup \{\varepsilon\}$, $Y \in V^*$ означает, что в текущем состоянии автомата k с элементом A в вершине стека при чтении символа a осуществляется переход в состояние k' и в стек заносится Y.

Конфигурация магазинного автомата в любой момент времени описывается элементом множества $K \times V^*$, т. е. парой, первый элемент которой задает текущее состояние автомата, а второй элемент – текущее содержимое стека. Тогда при вычислении

функции $\delta(k, A, a) = (k', Y)$ при условии ввода входного символа a автомат переходит из конфигурации $c = (k, Ax)$, $k \in K$, $A \in V$, $x \in V^*$ в конфигурацию $c' = (k', Yx)$, $k' \in K$, $Y \in V^*$.

Функция $\delta(k, A, \varepsilon) \neq \varnothing$ вызывает переход из одной конфигурации в другую без ввода входного символа. Такие перемещения автомата называются ε-*перемещениями*. Пустота стека является достаточным условием невозможности каких-либо перемещений автомата, кроме ε-перемещений.

Говорят, что строка принимается магазинным автоматом, если в результате ввода этой строки автомат перейдет из исходной конфигурации в конфигурацию с конечным состоянием.

Существуют и другие эквивалентные определения магазинного автомата:

$M' = (K, T, V, \delta, k_0, z_0, k_f)$, где k_f – единственное конечное состояние, при переходе в которое (и только в этом случае) стек автомата очищается.

$M'' = (K, T, V, \delta, k_0, z_0)$, т. е. не определено множество конечных состояний. В этом случае в результате ввода строки стек становится пустым.

Магазинный автомат называется *детерминированным*, если выполняются следующие условия:

а) функция вида $\delta(k, A, a)$ имеет не более одного элемента;

б) функция вида $\delta(k, A, \varepsilon)$ имеет не более одного элемента;

в) если $\delta(k, A, \varepsilon) \neq \varnothing$, то $\delta(k, A, a) = \varnothing$ для любого $a \in T$, т. е. если из некоторой конфигурации можно осуществить хотя бы одно ε-перемещение, то оно является единственным перемещением, которое можно осуществить из этой конфигурации.

Другими словами, если из любой конфигурации возможно единственное перемещение, магазинный автомат является детерминированным, в противном случае – *недетерминированным*.

Рассмотрим вопросы построения магазинного автомата, распознающего КС-язык, заданный КС-грамматикой. Пусть язык порождается грамматикой $G = (V_T, V_N, P, S)$. Магазинный автомат $M = (K, T, V, \delta, k_0, z_0, F)$, принимающий данный язык, определяется следующим образом:

$K = \{k_1, k_2, k_3\}$;

$T = V_T$;

$V = V_N \cup V_T \cup \{\#\}$, $\# \notin V_N \cup V_T$ – указатель дна стека;

$k_0 = k_1$;

$z_0 = \#$;

$F = \{k_3\}$,

т. е. $M = (K, T, V, \delta, k_1, \#, \{k_3\})$, где магазинная функция δ имеет следующие значения:

$\delta(k_1, \#, \varepsilon) = \{(k_2, S\#)\}$;

$\delta(k_2, A, \varepsilon) = \{(k_2, \alpha) \mid A \to \alpha$ – продукция из $P\}$ для любого $A \in V_N$;

$\delta(k_2, a, a) = \{(k_2, \varepsilon)\}$ для любого $a \in V_T$;

$\delta(k_2, \#, \varepsilon) = \{(k_3, \varepsilon)\}$.

Такой магазинный автомат эмулирует левосторонний вывод с помощью операций со стеком. При каждом перемещении, выполняемом автоматом, из стека извлекается один символ. Если извлеченный символ оказывается нетерминалом, то ему в соответствие ставится продукция, правая часть которой в таком случае заносится в стек. Если же извлеченный из стека символ оказывается терминалом, то он используется в качестве входного символа и, следовательно, определяет следующее перемещение автомата.

Например, недетерминированный магазинный автомат для грамматики

$E \to E + T \mid T$,

$T \to T \times F \mid F$,

$F \to (E) \mid i$

будет иметь следующие функции переходов:

$\delta(k_1, \#, \varepsilon) = \{(k_2, E\#)\}$;

$\delta(k_2, E, \varepsilon) = \{(k_2, E + T), (k_2, T)\}$;

$\delta(k_2, T, \varepsilon) = \{(k_2, T \times F), (k_2, F)\}$;

$\delta(k_2, F, \varepsilon) = \{(k_2, (E)), (k_2, i)\}$;

$\delta(k_2, i, i) = \{(k_2, \varepsilon)\}$;

$\delta(k_2, +, +) = \{(k_2, \varepsilon)\}$;

$\delta(k_2, \times, \times) = \{(k_2, \varepsilon)\}$;

$\delta(k_2, (, ( ) = \{(k_2, \varepsilon)\}$;

$\delta(k_2,),)) = \{(k_2, \varepsilon)\}$;

$\delta(k_2, \#, \varepsilon) = \{(k_3, \varepsilon)\}$.

Язык, принимаемый детерминированным магазинным автоматом, называется *детерминированным* языком. Не всякий КС-

язык является детерминированным, и такие языки не могут анализироваться детерминированным образом. Тем не менее детерминированные языки составляют очень важный класс языков, поскольку для них значительно упрощается решение задачи анализа. Большинство языков программирования являются детерминированными или почти таковыми. Некоторые языки можно разбирать детерминированно с помощью только одного из методов грамматического разбора.

Из недетерминированности автомата следует, что построенный на его основе синтаксический анализатор КС-языка в процессе функционирования может осуществлять возврат к предыдущему состоянию. Это означает, что, обнаружив ошибку выбора, необходимо вернуться к моменту осуществления выбора, вновь осуществить выбор и выполнить другое перемещение. Однако, если на порождающую язык грамматику наложить определенные ограничения, то можно построить эффективный детерминированный синтаксический анализатор для такого языка.

Нисходящие методы синтаксического анализа основаны на просмотре входной строки и построении синтаксического дерева, начиная с начального нетерминала грамматики. Дерево строится до тех пор, пока не получится анализируемая строка терминалов. Если такую строку удается получить, то анализируемая строка принадлежит языку, если нет – не принадлежит. Этот процесс равносилен процессу построения соответствующей схемы вывода анализируемой строки.

Во все рассматриваемые далее грамматики введем специальный символ $\perp$, принадлежащий множеству терминалов и обозначающий конец вводимой строки (маркер конца ввода). Такие грамматики можно назвать *пополненными*. Даже если маркер явно не указан в грамматике, будем подразумевать, что всегда имеется продукция вида $S' \to S\perp$, где S' и S – начальные нетерминалы соответственно пополненной и исходной грамматик.

4.2. *LL*(*k*)-грамматики

Рассмотрим так называемый *LL*-разбор. Для успешного *LL*-разбора строк некоторого КС-языка на порождающую его грамматику должны быть наложены строгие ограничения. Практически это означает, что методом *LL*-разбора можно воспользоваться лишь на подмножестве класса детерминированных языков.

Рассмотрим грамматику с продукциями

$S \to A\perp$,

$A \to aFaB \mid bFbB$,

$F \to b \mid ba$,

$B \to b \mid bB$.

Построим дерево нисходящего синтаксического разбора строки $abaabb\perp$ (рис. 3). Корень дерева помечен символом S и, следовательно, единственной продукцией грамматики, которая может быть использована в данном случае, является продукция $S \to A\perp$. Перейдем теперь к вершине A. Можно воспользоваться продукциями $A \to aFaB$ и $A \to bFbB$. Поскольку рассматриваемая строка $abaabb\perp$ начинается с символа a, воспользуемся первой из этих продукций, т. е. выбирается та продукция, правая часть которой начинается с очередного введенного входного символа. Итак, один из листьев дерева уже помечен символом a. Далее необходимо показать, что строка $FaB\perp$ может породить строку $baabb\perp$. Рассмотрим для этого продукции $F \to b \mid ba$. На этот раз знать следующий введенный символ и даже два следующих введенных символа входной строки недостаточно для выбора одной из этих продукций. Однако, если станут известны три очередных символа входной строки и если это символы bab, то следует воспользоваться первой продукцией, а если это символы baa, то второй продукцией. В данном случае очередные символы входной строки – это символы baa, поэтому следует выбрать продукцию $F \to ba$. Теперь необходимо убедиться, что из нетерминала B можно породить строку bb. Для этого надо рассмотреть продукции $B \to b \mid bB$. Опять знания следующего символа b недостаточно, но если известны два очередных символа и это символы $b\perp$, то следует использовать продукцию $B \to b$, а если это символы bb, – продукцию $B \to bB$. В данном случае очередные символы – это символы bb, поэтому используем продукцию $B \to bB$. Затем, согласно двум очередным символам входной строки, применим продукцию $B \to b$. В результате будет получена левосторонняя схема вывода

$S \Rightarrow A\perp \Rightarrow aFaB\perp \Rightarrow abaaB\perp \Rightarrow abaabB\perp \Rightarrow abaabb\perp$.

Эта грамматика является примером $LL(3)$-грамматики, т. е. для однозначного определения дерева вывода достаточно знать

не более трех очередных символов входной строки на каждом этапе построения дерева.

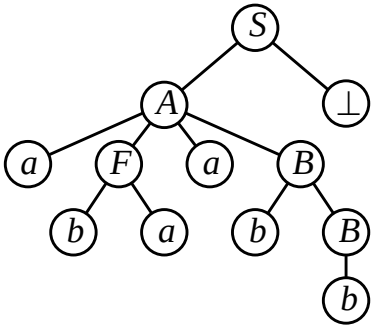

Рис. 3

В общем случае *LL*(*k*)-грамматикой называют такую грамматику, что для любой ее сентенциальной формы wAy, $w \in V_T^*$, $A \in V_N$, $y \in (V_N \cup V_T)^*$, полученной в результате некоторого левостороннего вывода, для однозначного выбора продукции, имеющей в левой части символ A, достаточно знать k очередных символов входной строки. Аббревиатура *LL* означает "левосторонний ввод – левосторонний вывод".

Чем больше k, тем больший класс языков может быть представлен *LL*(*k*)-грамматикой. Однако на практике наибольшее применение имеют *LL*(1)-грамматики, для которых детерминированный распознаватель работает, анализируя по одному входному символу, расположенному в текущей позиции.

4.3. *LL*(1)-грамматики

Прежде чем определить *LL*(1)-грамматику, рассмотрим некоторые подклассы *LL*(1)-грамматик и вопросы их разбора.

4.3.1. Разделенные грамматики

Разделенная, или *простая,* грамматика (*s-грамматика*) представляет собой грамматику, в которой правая часть каждой продукции начинается с терминального символа, за терминалом могут следовать нетерминалы и/или терминалы; если несколько продукций имеют одинаковые левые части, соответствующие правые части начинаются с разных терминалов.

Эти условия позволяют написать детерминированный нисходящий анализатор, т. к. при выводе строки языка всегда можно сделать выбор между альтернативными продукциями для самого левого нетерминала в сентенциальной форме, предварительно исследовав один следующий символ входной строки.

Например, грамматика с продукциями

$S \to pA \mid qB,$

$A \to a \mid cAd,$

$B \to b \mid cBg$

представляет собой *s*-грамматику.

4.3.2. Слаборазделенные грамматики

Пусть дана следующая грамматика:

$S \to pA \mid qBA,$

$A \to a \mid bAd,$

$B \to \varepsilon \mid cBg.$

Эта грамматика не принадлежит классу *s*-грамматик, поскольку содержит ε-продукцию $B \to \varepsilon$, правая часть которой не начинается с терминала. Выбрать при разборе продукцию по левому терминалу правой части не удается.

Пусть $FOLLOW(X)$ – множество терминалов, которые могут следовать непосредственно за нетерминалом X в какой-либо сентенциальной форме, выводимой из начального нетерминала. Например, для приведенной грамматики $FOLLOW(B) = \{a, b, g\}$.

Критерием выбора между альтернативными продукциями является *множество направляющих символов* (*множество выбора*) *DS*, определяемое следующим образом:

Если продукция имеет вид $A \to b\alpha$, где $A \in V_N$, $b \in V_T$, $\alpha \in (V_T \cup V_N)^*$, то $DS(A \to b\alpha) = \{b\}$. Если продукция имеет вид $A \to \varepsilon$, то $DS(A \to \varepsilon) = FOLLOW(A)$.

Контекстно-свободная грамматика называется *слаборазделенной* (*q-грамматикой*) при условии:

а) правая часть каждой продукции либо начинается с терминала, либо представляет собой ε;

б) множества направляющих символов продукций с одинаковой левой частью не пересекаются.

Таким образом, приведенная выше грамматика относится к классу *q*-грамматик, т. к. множества *DS* альтернативных продук-

ций не пересекаются. Это позволяет при разборе строки детерминированно выбирать нужную продукцию из альтернативных.

Процесс вывода строки $qbad\perp$ соответствует следующей левосторонней схеме вывода:

$S\perp \Rightarrow qBA\perp \Rightarrow qA\perp \Rightarrow qbAd\perp \Rightarrow qbad\perp$.

На втором шаге вывода, поскольку $b \in DS(B \to \varepsilon) = \{a, b, g\}$, применена продукция $B \to \varepsilon$.

4.3.3. LL(1)-грамматики

LL(1)-грамматика является обобщением *q*-грамматики, принцип обобщения позволяет строить нисходящие детерминированные анализаторы.

Пусть $G = (V_N, V_T, P, S)$ произвольная КС-грамматика. Определим следующие функции:

EMPTY(*X*) = **true**, если *X* является ε-порождающим нетерминалом, и *EMPTY*(*X*) = **false** в противном случае. Расширим понятие функции *EMPTY* на строку:

EMPTY(ε) = **true**,

EMPTY($X\alpha$) = *EMPTY*(*X*) **and** *EMPTY*(α).

Эта функция определяет, может данная строка генерировать пустую строку или нет.

$$FIRST(X) = \{a \mid X \overset{*}{\Rightarrow} a\alpha\},\ X \in (V_T \cup V_N),\ a \in V_T,\ \alpha \in (V_T \cup V_N)^*,$$

т. е. функция *FIRST*(*X*) определяет множество терминалов, с которых может начинаться строка, выводимая из символа *X*. Расширим понятие функции *FIRST* на строку символов $\alpha = X_1X_2...X_n$, где $X_i \in V_N \cup V_T$, $1 \le i \le n$:

$$FIRST(X_1X_2...X_n) = \bigcup_{i=1}^{n} FIRST(X_i) \mid EMPTY(X_1X_2...X_{i-1}),$$

т. е. определяет множество терминалов, с которых может начинаться строка, выводимая из строки $\alpha = X_1X_2...X_n$, учитывая наличие ε-порождающих нетерминалов. Ясно, что *FIRST*(ε) = ∅.

Аргументом следующей функции является нетерминал $X \in V_N$:

$$FOLLOW(X) = \{a \mid S \overset{*}{\Rightarrow} \alpha Xa\beta\},\ a \in V_T,\ \alpha \in V_T^*,\ \beta \in (V_T \cup V_N)^*.$$

Данная функция (как и в *q*-грамматике) определяет множество терминалов, которые могут следовать непосредственно за

нетерминалом X в какой-либо сентенциальной форме, выводимой из начального нетерминала S.

Тогда множество DS направляющих символов продукций $LL(1)$-грамматики определяется следующим образом:

$DS(A \to \alpha) = FIRST(\alpha) \cup FOLLOW(A)$, если $EMPTY(\alpha) =$ **true**,

$DS(A \to \alpha) = FIRST(\alpha)$ в противном случае.

Другими словами, если правая часть продукции может генерировать пустую строку, то к элементам множества $FIRST(\alpha)$ необходимо добавить элементы множества $FOLLOW(A)$ для нетерминала из левой части продукции.

Контекстно-свободная грамматика $G = (V_N, V_T, P, S)$ называется $LL(1)$-*грамматикой* тогда и только тогда, когда для любой пары несовпадающих продукций вида $A \to \alpha$ и $A \to \beta$, имеющих одинаковые левые части, справедливо утверждение, что

$$DS(A \to \alpha) \cap DS(A \to \beta) = \varnothing,$$

т. е. множества направляющих символов, соответствующих правым частям альтернативных продукций, не пересекаются.

4.4. Алгоритм распознавания $LL(1)$-грамматик

Чтобы определить, относится ли заданная КС-грамматика к классу $LL(1)$-грамматик, необходимо определить множества направляющих символов, вычислив предварительно функции *EMPTY*, *FIRST* и *FOLLOW*, и установить, пересекаются данные множества для продукций с одинаковой левой частью или нет. Рассмотрим процесс вычисления функций; проиллюстрируем его на примере грамматики с продукциями (в предположении, что имеется продукция $S' \to S\perp$ с маркером конца вводимой строки):

$S \to ABC$, $\quad E \to aa \mid \varepsilon$,
$A \to DE$, $\quad F \to HK$,
$B \to FG$, $\quad G \to bb$,
$C \to \varepsilon$, $\quad H \to cc$,
$D \to a \mid \varepsilon$, $\quad K \to dd$.

4.4.1. Вычисление функции EMPTY

Вычислить функцию *EMPTY*, т. е. определить множество всех ε-порождающих нетерминалов, можно в соответствии со следующей процедурой.

Определить вектор, элементы которого соответствуют нетерминалам грамматики. Элемент вектора может принимать одно из следующих трех значений: *T* (нетерминал является ε-порождающим), *F* (нетерминал не ε-порождающий) или *U* (значение элемента не определено). Вначале все элементы вектора имеют значение *U*. В процессе анализа продукции грамматики просматриваются до тех пор, пока все элементы вектора не примут значение *T* или *F*.

При первом просмотре исключаются все продукции, содержащие в правой части хотя бы один терминал. Если это приведет к исключению всех продукций для какого-либо нетерминала левой части, соответствующему элементу вектора присваивается значение *F*. Элементам вектора, соответствующим нетерминалам левых частей всех ε-продукций, присваивается значение *T*, и все продукции для этих нетерминалов исключаются из грамматики.

Если в векторе еще имеются элементы с неопределенным значением *U*, то требуются дополнительные просмотры и выполняются следующие действия:

1. Каждая продукция, имеющая такой символ в правой части, который не может генерировать пустую строку (определяется по значению соответствующего элемента вектора), исключается из грамматики. В случае, когда для нетерминала в левой части исключенной продукции не существует других продукций, значение элемента вектора, соответствующего этому нетерминалу, устанавливается в *F*.

2. Каждый нетерминал в правой части продукции, который может генерировать пустую строку, исключается из продукции. В случае, если правая часть продукции становится пустой, элементу вектора, соответствующего нетерминалу в левой части, присваивается значение *T*, и все продукции для этого нетерминала исключаются из грамматики.

Этот процесс продолжается до тех пор, пока за полный просмотр грамматики не изменяются значения элементов вектора, т. е. все элементы не примут значение *T* или *F*. Если в результате для некоторого нетерминала *A* соответствующий элемент вектора примет значение *T*, то этот нетерминал *A* является ε-порождающим, т. е. *EMPTY*(*A*) = **true**.

Рассмотрим этот процесс для нашей грамматики. После первого прохода вектор будет иметь вид

S	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*K*
U	*U*	*U*	*T*	*T*	*T*	*U*	*F*	*F*	*F'*

а в грамматике останутся продукции

$S \to ABC$,

$A \to DE$,

$B \to FG$,

$F \to HK$.

Вектор имеет элементы с неопределенными значениями, поэтому требуется второй проход. У продукции $S \to ABC$ нетерминал *C* в правой части является ε-порождающим, поэтому он исключается из продукции. У продукции $A \to DE$ нетерминалы в правой части являются ε-порождающими, т. е. правая часть продукции может генерировать пустую строку, поэтому элементу вектора, соответствующего нетерминалу *A*, присваивается значение *T*. У продукции $B \to FG$ нетерминал *G* в правой части не является ε-порождающим, т. е. правая часть продукции не может генерировать пустую строку, поэтому элементу вектора, соответствующего нетерминалу *B*, присваивается значение *F*. У продукции $F \to HK$ нетерминалы в правой части не являются ε-порождающими, поэтому элементу вектора, соответствующего нетерминалу *F*, присваивается значение *F*. В результате второго прохода получается вектор

S	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*K*
U	*T*	*F*	*T*	*T*	*T*	*F*	*F*	*F*	*F'*

а в множестве продукций остается продукция $S \to AB$. Третий проход, поскольку в этой продукции нетерминал *B* не может генерировать пустую строку, устанавливает значение вектора для S в *F* и завершает формирование вектора. Таким образом, нетерминалы *A*, *C*, *D* и *E* являются ε-порождающими, т. е. значение функции *EMPTY* для них равно **true**, а для остальных нетерминалов – **false**.

4.4.2. Вычисление функции FIRST

Формальный процесс вычисления функции *FIRST* можно представить такой последовательностью действий:

1. Построить отношение <НАЧИНАЕТСЯ_ПРЯМО_С>.

A <НАЧИНАЕТСЯ_ПРЯМО_С> *B*, если в грамматике существует продукция вида $A \to \alpha B \beta$, где α – ε-порождающая подстрока, β – произвольная подстрока, $B \in V_N \cup V_T$.

Для нашего примера данное отношение представлено на рис. 4. Здесь *S* <НАЧИНАЕТСЯ_ПРЯМО_С> *B*, поскольку в продукции $S \to ABC$ нетерминал *A* является ε-порождающим. Аналогичная ситуация возникает и для нетерминала *A*.

	S	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*K*	*a*	*b*	*c*	*d*	⊥
S		1	1												
A					1	1									
B							1								
C															
D											1				
E											1				
F									1						
G												1			
H													1		
K														1	

Рис. 4

2. Вычислить отношение <НАЧИНАЕТСЯ_С> как рефлексивно-транзитивное замыкание отношения <НАЧИНАЕТСЯ_ПРЯМО_С>.

Результат вычисления данного отношения для нашего примера представлен на рис. 5.

3. Вычислить значение функции *FIRST*(*A*) для каждого нетерминала. Это множество таких терминалов *a*, для которых выполняется отношение *A* <НАЧИНАЕТСЯ_С> *a*.

Для нашего примера

FIRST(*S*) = {*a*, *c*}, *FIRST*(*E*) = {*a*},
FIRST(*A*) = {*a*}, *FIRST*(*F*) = {*c*},
FIRST(*B*) = {*c*}, *FIRST*(*G*) = {*b*},
FIRST(*C*) = ∅, *FIRST*(*H*) = {*c*},
FIRST(*D*) = {*a*}, *FIRST*(*K*) = {*d*}.

4. Вычислить значение функции *FIRST* для правой части каждой продукции грамматики.

	S	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*K*	*a*	*b*	*c*	*d*	⊥
S	1	1	1		1	1	1		1		1		1		
A		1			1	1					1				
B			1				1		1				1		
C				1											
D					1						1				
E						1					1				
F							1		1				1		
G								1				1			
H									1				1		
K										1				1	
a											1				
b												1			
c													1		
d														1	
⊥															1

Рис. 5

Для нашего примера
$FIRST(ABC) = FIRST(A) \cup FIRST(B) = \{a, c\}$,
$FIRST(DE) = FIRST(D) \cup FIRST(E) = \{a\}$,
$FIRST(FG) = FIRST(F) = \{c\}$,
$FIRST(a) = \{a\}$,
$FIRST(aa) = \{a\}$,
$FIRST(HK) = FIRST(H) = \{c\}$,
$FIRST(bb) = \{b\}$,
$FIRST(cc) = \{c\}$,
$FIRST(dd) = \{d\}$,
$FIRST(\varepsilon) = \varnothing$.

4.4.3. Вычисление функции FOLLOW

Формальный процесс вычисления функции *FOLLOW* можно представить такой последовательностью действий:

1. Построить отношение <ПРЯМО_ПЕРЕД>.

A <ПРЯМО_ПЕРЕД> B, если в грамматике существует продукция вида $D \to \alpha A \beta B \gamma$, где β – ε-порождающая подстрока; α, γ – произвольные подстроки; $A \in V_N$; $B \in (V_N \cup V_T)$. Для нашего примера данное отношение представлено на рис. 6.

	S	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*K*	*a*	*b*	*c*	*d*	⊥
S															1
A			1												
B				1											
C															
D						1									
E															
F								1							
G															
H										1					
K															

Рис. 6

2. Построить отношение <ПРЯМО_НА_КОНЦЕ>.

A <ПРЯМО_НА_КОНЦЕ> B, если в грамматике имеется продукция вида $B \to \alpha A \beta$, где $A \in V_N$; β – ε-порождающая подстрока; α – произвольная подстрока. Это отношение определяется только для нетерминалов.

Для нашего примера данное отношение представлено на рис. 7. Здесь *B* <ПРЯМО_НА_КОНЦЕ> *S*, поскольку в продукции $S \to ABC$ нетерминал *C* является ε-порождающим; аналогичная ситуация для отношения *D* <ПРЯМО_НА_КОНЦЕ> *A*, поскольку в продукции $A \to DE$ нетерминал *E* является ε-порождающим.

	S	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*K*
S										
A										
B	1									
C	1									
D		1								
E		1								
F										
G			1							
H										
K							1			

Рис. 7

3. Вычислить отношение <НА_КОНЦЕ> как рефлексивно-транзитивное замыкание отношения <ПРЯМО_НА_КОНЦЕ>.

Для нашего примера результат вычисления данного отношения представлен на рис. 8.

	S	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*K*
S	1									
A		1								
B	1		1							
C	1			1						
D		1			1					
E		1				1				
F							1			
G	1		1					1		
H									1	
K							1			1

Рис. 8

4. Вычислить отношение <ПЕРЕД>.

A <ПЕРЕД> *B*, когда из начального нетерминала *S* можно вывести сентенциальную форму, в которой за вхождением символа *A* сразу же следует вхождение символа *B*.

Если *A* <ПЕРЕД> *B*, то должны существовать такие символы *X* и *Y*, что

A <НА_КОНЦЕ> *X* <ПРЯМО_ПЕРЕД> *Y* <НАЧИНАЕТСЯ_С> *B*.

Таким образом, отношение <ПЕРЕД> является произведением отношений:

<НА_КОНЦЕ> · <ПРЯМО_ПЕРЕД> · <НАЧИНАЕТСЯ_С>.

Для нашего примера результат вычисления отношения <ПЕРЕД> представлен на рис. 9.

5. Вычислить значение функции *FOLLOW*(*A*) для каждого ε-порождающего нетерминала. Это множество таких терминалов *a*, для которых выполняется отношение *A* <ПЕРЕД> *a*.

Для нашего примера имеем следующие значения:

FOLLOW(*A*) = {*c*},

FOLLOW(*C*) = {⊥},

FOLLOW(*D*) = {*a*, *c*},

FOLLOW(*E*) = {*c*}.

	S	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*K*	*a*	*b*	*c*	*d*	⊥
S															1
A			1				1		1				1		
B				1											1
C															1
D			1			1	1		1		1		1		
E			1				1		1				1		
F								1				1			
G				1											1
H										1				1	
K								1				1			

Рис. 9

4.4.4. Вычисление множеств направляющих символов

На последнем этапе, когда вычислены все необходимые функции, легко определяются множества направляющих символов *DS* продукций грамматики. Эти множества достаточно определить только для продукций с одинаковой левой частью. Тогда, проверив, пересекаются множества или нет, можно ответить на вопрос, является данная грамматика *LL*(1)-грамматикой или нет.

Для нашего примера имеем следующие множества направляющих символов (с целью обучения показаны для всех продукций грамматики):

$DS(S \to ABC) = FIRST(ABC) = \{a, c\}$,
$DS(A \to DE) = FIRST(DE) \cup FOLLOW(A) = \{a, c\}$,
$DS(B \to FG) = FIRST(FG) = \{c\}$,
$DS(C \to \varepsilon) = FOLLOW(C) = \{\perp\}$,
$DS(D \to a) = FIRST(a) = \{a\}$,
$DS(D \to \varepsilon) = FOLLOW(D) = \{a, c\}$,
$DS(E \to aa) = FIRST(aa) = \{a\}$,
$DS(E \to \varepsilon) = FOLLOW(E) = \{c\}$,
$DS(F \to HK) = FIRST(HK) = \{c\}$,
$DS(G \to bb) = FIRST(bb) = \{b\}$,
$DS(H \to cc) = FIRST(cc) = \{c\}$,
$DS(K \to dd) = FIRST(dd) = \{d\}$.

Таким образом, грамматика не является *LL*(1)-грамматикой, поскольку для продукций $D \to a$ и $D \to \varepsilon$ множества $DS(D \to a) = \{a\}$ и $DS(D \to \varepsilon) = \{a, c\}$ пересекаются.

4.5. Приемы преобразования грамматик в *LL*(1)-форму

Не существует полностью универсального автоматического процесса преобразования грамматик в LL(1)-форму. Отсутствие общего решения проблемы не означает невозможности ее решения для частных случаев.

Прежде всего следует заметить, что грамматика, содержащая левую рекурсию, не является *LL*(1)-грамматикой. Рассмотрим продукции

$A \to A\alpha$ (леворекурсивная продукция по A),

$A \to a$.

$DS(A \to A\alpha) = \{a\}$ и $DS(A \to a) = \{a\}$, т. е. множества направляющих символов пересекаются.

По тем же причинам грамматика, содержащая левый рекурсивный цикл, не может быть *LL*(1)-грамматикой.

Как уже отмечалось, левую рекурсию всегда можно исключить из грамматики, преобразовав ее в правую рекурсию, которая не вызывает никаких проблем для нисходящего разбора.

Другим преобразованием, которое часто используется для получения *LL*(1)-грамматик, является факторизация. Рассмотрим ее на примере следующей грамматики:

$S \to aSb \mid aSc \mid \varepsilon$.

Определим множества направляющих символов (учитывая, что грамматика пополнена маркером конца строки $\perp$, т. е. в предположении, что в грамматике имеется продукция $S' \to S\perp$):

$DS(S \to aSb) = \{a\}$,

$DS(S \to aSc) = \{a\}$,

$DS(S \to \varepsilon) = FOLLOW(S) = \{b, c, \perp\}$.

Направляющий символ a является общим для двух первых продукций, т. е. это не *LL*(1)-грамматика. Применим правило факторизации. В результате получится *LL*(1)-грамматика

$S \to aSX \mid \varepsilon$,

$X \to b \mid c$.

Таким образом, факторизация как бы "выносит за скобки" направляющие символы.

Однако процесс факторизации нельзя распространить на общий случай. Следующий пример показывает, что может произойти. Рассмотрим грамматику

$S \to Ac \mid Bd,$

$A \to eAf \mid a,$

$B \to eBg \mid b.$

Первые две продукции с символом S в левой части в своих множествах направляющих символов содержат символ e. Для проведения факторизации предварительно выполним замену вхождений нетерминалов A и B, чтобы направляющие символы явно присутствовали в этих продукциях:

$S \to eAfc \mid ac \mid eBgd \mid bd.$

Выполняя факторизацию, эти продукции можно заменить следующими:

$S \to ac \mid bd \mid eS_1,$

$S_1 \to Afc \mid Bgd.$

Продукции для S_1 аналогичны первоначальным продукциям для S и имеют пересекающиеся множества направляющих символов. Можно повторить преобразование этих продукций, как это было сделано с продукциями для S:

$S_1 \to eAffc \mid afc \mid eBggd \mid bgd.$

В результате факторизации получим

$S_1 \to afc \mid bgd \mid eS_2,$

$S_2 \to Affc \mid Bggd.$

Продукции для S_2 аналогичны продукциям для S_1 и S, но длиннее их, и теперь очевидно, что этот процесс бесконечный.

Это означает, что все попытки преобразовать грамматику в $LL(1)$-форму не всегда приводят к результату. Это является прямым следствием того, что $LL(1)$-языки являются только подклассом более широкого класса языков, допускающих детерминированный разбор.

Для иллюстрации преобразуем в $LL(1)$-форму следующую грамматику:

$E \to E + T \mid T,$

$T \to T \times F \mid F,$

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

Она не является $LL(1)$-грамматикой, поскольку содержит леворекурсивные продукции. После устранения левой рекурсии получаются продукции

$E \to T \mid TE'$,
$E' \to + T \mid + TE'$,
$T \to F \mid FT'$,
$T' \to \times F \mid \times FT'$,
$F \to (E) \mid i$.

Применив факторизацию, получаем

$E \to TX$,
$X \to E' \mid \varepsilon$,
$E' \to + TX$,
$T \to FY$,
$Y \to T' \mid \varepsilon$,
$T' \to \times FY$,
$F \to (E) \mid i$.

Выполнив одиночные замены для нетерминалов E' и T', получаем грамматику

$E \to TX$,
$X \to + TX \mid \varepsilon$,
$T \to FY$,
$Y \to \times FY \mid \varepsilon$,
$F \to (E) \mid i$.

Это $LL(1)$-грамматика. Покажем это, определив множества направляющих символов для продукций с одинаковыми левыми частями:

$DS(X \to + TX) = \{+\}$,
$\mathrm{DS}(X \to \varepsilon) = \{), \perp\}$,
$DS(Y \to \times FY) = \{\times\}$,
$\mathrm{DS}(Y \to \varepsilon) = \{+,), \perp\}$,
$DS(F \to (E)) = \{(\}$,
$\mathrm{DS}(F \to i) = \{i\}$.

Множества направляющих символов всех пар продукций с одинаковыми левыми частями не пересекаются.

4.6. Рекурсивный спуск

Метод рекурсивного спуска – хорошо известный и легко реализуемый детерминированный метод нисходящего разбора для *LL*(*k*)-грамматик.

В простейшей форме метод рекурсивного спуска предоставляет удобную возможность построения синтаксического анализатора языка, порожденного *LL*(1)-грамматикой. Такой синтаксический анализатор каждому символу из множества $V_T \cup V_N$ ставит в соответствие единственную процедуру, с помощью которой для любой строки языка можно однозначно определить, выводима она из этого символа или нет.

Если $a \in V_T$, то соответствующая процедура (обозначим *proc_a*) обеспечивает ввод следующего входного символа и сравнение его с символом *a*. Если $A \in V_N$ и продукции грамматики с нетерминалом *A* в левой части имеют вид $A \to \alpha_1 | \alpha_2 | ... | \alpha_n$, то, сравнив очередной входной символ с элементами множеств направляющих символов, можно определить, какой продукцией нужно воспользоваться для осуществления следующего шага вывода. Если с помощью процедуры *proc_A* и в зависимости от очередного входного символа выбрана продукция $A \to \alpha_i$, и $\alpha_i = X_1X_2...X_m$, $X_j \in V_T \cup V_N$, $1 \le j \le m$, то можно говорить о последовательности вызовов процедур *proc_*X_1; *proc_*X_2; ...; *proc_*X_m.

В качестве примера рассмотрим LL(1)-грамматику со следующими продукциями (для каждой продукции справа указаны множества направляющих символов)

$S \to aAB$ $\{a\}$,
$S \to bS$ $\{b\}$,
$A \to aAb$ $\{a\}$,
$A \to bBc$ $\{b\}$,
$B \to AB$ $\{a, b\}$,
$B \to c$ $\{c\}$.

Пусть процедура *read*(*sym*) считывает из входной сроки очередной символ и присваивает его значение переменной *sym*, процедура *error* каким-либо образом обрабатывает синтаксическую ошибку, процедура *stop* завершает синтаксический разбор. Тогда синтаксический анализатор можно представить множеством соответствующих процедур (рис. 10).

```
procedure proc_S
   case { sym = 'a': proc_a; proc_A; proc_B
        { sym = 'b': proc_b; proc_S
return
procedure proc_A
   case { sym = 'a': proc_a; proc_A; proc_b
        { sym = 'b': proc_b; proc_B; proc_c
return
procedure proc_B
   case { sym = 'a' or 'b': proc_A; proc_B
        { sym = 'c': proc_c
return
procedure proc_a
   if sym = 'a' then read(sym) else error
return
procedure proc_b
   if sym = 'b' then read(sym) else error
return
procedure proc_c
   if sym = 'c' then read(sym) else error
return
```

Рис. 10

Процесс разбора начинается со следующих операций:

read(*sym*); *proc_S*; **if** *sim* = ⊥ **then** *stop* **else** *error*.

С целью сокращения числа вызовов процедур можно такие процедуры поставить в соответствие только для нетерминалов, включая операции ввода и анализа терминалов в эти процедуры (рис. 11).

Для разбора предложений языка может потребоваться много рекурсивных вызовов процедур, соответствующих нетерминалам в грамматике. Если представить грамматику несколько иным способом, в ряде случаев рекурсию можно заменить итерацией. Например, пусть дана грамматика со следующими продукциями:

$S \to cAdBe$,

$A \to afA \mid a$,

$B \to bfB \mid b$.

Ее можно представить следующим образом (используя нотацию регулярных выражений):

$S \to cAdBe$,
$A \to a(fa)^*$,
$B \to b(fb)^*$.

Тогда процедуры для нетерминалов A и B можно представить так, как показано на рис. 12. Такая замена рекурсии итерацией обычно делает анализатор более эффективным.

```
procedure proc_S
   if sym = 'a'
      then read(sym); proc_A; proc_B
      elseif sym = 'b'
               then read(sym); proc_S
               else error
return
procedure proc_A
   if sym = 'a'
      then read(sym); proc_A; if sym = 'b'
                                 then read(sym)
                                 else error
      elseif sym = 'b'
         then read(sym); proc_B; if sym = 'c'
                                    then read(sym)
                                    else error
         else error
return
procedure proc_B
   if sym = 'c' then read(sym) else proc_A; proc_B
return
```

Рис. 11

Метод рекурсивного спуска является одним из наиболее эффективных способов создания компиляторов. Преимущества написания рекурсивного нисходящего анализатора очевидны. Основные из них – это скорость написания анализатора на основании соответствующей грамматики. Другое преимущество заключается в соответствии между грамматикой и анализатором, благодаря которому увеличивается вероятность того, что анализатор окажется правильным, или, по крайней мере, того, что ошибки будут носить простой характер.

Недостатки этого метода, хотя и менее очевидны, но не менее реальны. Из-за большого числа вызовов процедур во время

синтаксического анализа анализатор становится относительно медленным. Кроме того, он может быть довольно большим по сравнению с анализаторами, основанными на табличных методах разбора. Несмотря на то, что данный метод способствует включению в анализатор действий по генерированию кода, это неизбежно приводит к смешиванию различных фаз компиляции. Это снижает надежность компиляторов или усложняет обращение с ними и привносит в анализатор зависимость от машины.

```
procedure proc_A
   if sym ≠ 'a' then error
   read(sym)
   while sym = 'f' do { read(sym)
                        if sym ≠ 'a' then error
                        read(sym)
return
procedure proc_B
   if sym ≠ 'b' then error
   read(sym)
   while sym = 'f' do { read(sym)
                        if sym ≠ 'b' then error
                        read(sym)
return
```

Рис. 12

4.7. *LL*(1)-таблицы разбора

Табличные методы синтаксического анализа аналогичны рекурсивному спуску. Здесь исключаются многочисленные вызовы процедур благодаря представлению грамматики в табличном виде (таблицы разбора) и использованию независящего от анализируемого языка модуля компилятора, проводящего синтаксический разбор по таблице.

Основным достоинством табличных методов разбора является то, что модуль синтаксического анализатора можно применять многократно в компиляторах для различных языков, изменив только содержимое таблицы разбора. Процесс формирования таблиц разбора для *LL*(1)-грамматик обычно носит детерминированный характер. Поэтому этот процесс можно легко автоматизировать, написав программу для получения соответствую-

щей таблицы разбора. В результате сроки проектирования компиляторов существенно сокращаются.

Таблицы разбора организуются таким образом, что модуль синтаксического анализа компилятора всегда указывает на то место в синтаксисе, которое соответствует текущему входному символу. Модулю требуется стек для запоминания адресов возврата всякий раз, когда он обрабатывает новую порождающую продукцию, соответствующую какому-либо нетерминалу. Представление синтаксиса в таблице должно быть таким, чтобы обеспечить эффективность синтаксического анализатора в отношении скорости работы.

Возможны различные виды таблиц разбора, которые являются, по сути, различными формами представления магазинного автомата, принимающего данный контекстно-свободный язык. Рассмотрим один достаточно простой и понятный вид таблицы разбора для *LL*(1)-грамматик.

LL(1)-таблица разбора представляет собой набор строк. Каждая строка содержит поля (столбцы):

а) список терминалов – *terminals*,

б) поле переходов – *jump*,

в) поле приема – *accept*,

г) поле стека – *stack*,

д) поле возврата – *return*,

е) поле ошибки – *error*.

Область значений поля *jump* – неотрицательные целые числа (номера строк таблицы), а область значений полей *accept*, *stack*, *return* и *error* – {**true**, **false**}.

В таблице каждому шагу процесса разбора соответствует один элемент. Поэтому в нее включаются по одному элементу для каждой продукции грамматики (для нетерминала из левой части) и на каждый экземпляр терминала или нетерминала правой части продукции. Кроме того, в таблицу включаются элементы на каждую реализацию пустой строки в правой части продукции (для ε-продукций). Первая строка таблицы соответствует первой продукции с начальным нетерминалом в левой части. Для этого исходная *LL*(1)-грамматика представляется в виде специальной схемы, в которой каждому символу грамматики соответствует номер строки таблицы разбора. Очень важно, чтобы номера нетерминалов в левых частях альтернативных продукций

непосредственно следовали друг за другом. Это необходимо для того, чтобы в случае, если анализируемый символ не принадлежит множеству направляющих символов одной продукции, увеличением номера на единицу перейти к проверке следующей альтернативной продукции.

Правила заполнения таблицы заключаются в следующем.

1. Поле *terminals*. Если строка таблицы соответствует продукции (т. е. нетерминалу в левой части), в поле заносится множество направляющих символов данной продукции. Если строка соответствует нетерминалу из правой части продукции, в поле записывается множество терминалов, являющееся объединением множеств направляющих символов всех продукций с данным нетерминалом в левой части. Если строка соответствует терминалу, в поле заносится этот терминал. Если строка соответствует правой части ε-продукции, в поле заносится множество направляющих символов этой ε-продукции.

2. Поле *jump*. В поле *jump* записывается номер строки следующего элемента обработки.

3. Поле *accept*. Устанавливается значение **true**, если строка таблицы соответствует терминалу, в противном случае – **false.** Означает, что если просматриваемый символ входной строки совпадает с данным терминалом, то этот символ можно принять и перейти к анализу следующего входного символа.

4. Поле *stack*. Устанавливается значение **true**, если строка таблицы соответствует нетерминалу из правой части продукции за исключением случая, когда этот нетерминал является крайним правым символом продукции. Поле показывает, что необходимо занести в стек точку возврата (номер строки для символа, непосредственно следующего в продукции за данным нетерминалом), поскольку далее осуществляется переход к обработке первой продукции, относящейся к данному нетерминалу. Очевидно, что если нетерминал является крайним правым символом продукции, нет необходимости помещать в стек адрес возврата. Во всех остальных случаях поле принимает значение **false.**

5. Поле *return*. Устанавливается значение **true**, если строка таблицы соответствует крайнему правому терминалу продукции, в остальных случаях – **false**. Означает, что завершена обработка продукции и необходимо вернуться к точке возврата, сохранен-

ной в стеке. При этом значение поля *jump* во внимание не принимается и устанавливается нулевое значение.

6. Поле *error*. Если имеется несколько альтернативных продукций, например k, то в строках, соответствующих первым $k - 1$ продукциям, поле ошибки принимает значение **false**, для k-й продукции – **true**. Это связано с тем, что если входной символ не является направляющим, то это еще не означает ошибку, поскольку он может быть направляющим для какой-либо другой альтернативной продукции, проверяемой на следующем этапе. Если же анализируемый символ не будет направляющим ни для одной из альтернативных продукций (а это выяснится только после проверки последней из них), тогда можно констатировать наличие синтаксической ошибки. Во всех остальных случаях поле ошибки принимает значение **true.**

В процессе синтаксического анализа осуществляется ряд различных шагов:

1. Проверка предварительно просматриваемого символа с целью выяснения, не является ли данный символ направляющим для какой-либо конкретной правой части продукции. Если этот символ не направляющий и имеется альтернативная продукция, то она проверяется на следующем этапе.

2. Проверка терминала, появляющегося в правой части продукции.

3. Проверка нетерминала. Она заключается в проверке нахождения предварительно просматриваемого символа в одном из множеств направляющих символов для данного нетерминала, помещении в стек адреса возврата и переходу к первой продукции, относящейся к этому нетерминалу.

Модуль синтаксического анализатора обрабатывает элемент таблицы разбора и определяет следующий элемент для обработки. Поле перехода *jump* дает следующий элемент обработки, если значение поля возврата *return* не окажется равным **true**. В последнем случае адрес следующего элемента берется из стека (что соответствует концу продукции). Если же предварительно просматриваемый символ отсутствует в списке терминалов и значение поля ошибки окажется **false**, нужно обрабатывать следующий элемент таблицы с тем же предварительно просматриваемым символом.

Алгоритм работы модуля синтаксического анализа приведен на рис. 13. Здесь переменная *la* представляет собой логическое значение, которое определяет, надо ли считывать новый предварительно просматриваемый символ до обработки следующего элемента таблицы разбора. Например, *la* = **false**, когда предварительно просматриваемый символ не является направляющим для какой-либо конкретной продукции и требуется исследовать множество направляющих символов следующей альтернативной продукции. Если символ не содержится в текущем множестве направляющих символов и поле ошибки *error* будет **true**, то выдается сообщение о синтаксической ошибке. Если поле стека обрабатываемой *i*-й строки таблицы разбора (TP_i) имеет значение **true**, то до перехода к адресу, задаваемому полем перехода, в стек помещается адрес следующей строки таблицы.

```
i ← 1 {номер строки таблицы разбора}
S ⇐ 0 {стек}
la ← true
read(sym)
while sym ≠ ⊥ and i ≠ 0 do
  with TP_i do
    if sym ∈ terminals
      then
        la ← accept
        if return
          then i ⇐ S
          else if stack then S ⇐ i + 1
                        i ← jump
      else if error
             then syntax_error
             else i ← i + 1
                  la ← false
    if la then read(sym)
```

Рис. 13

Такой модуль синтаксического анализа совершенно не зависит от разбираемого языка и может использоваться в целом ряде компиляторов. Он относительно мал, т. к. большая часть объема памяти, занимаемого синтаксическим анализатором, приходится на таблицу разбора, размер которой пропорционален размеру грамматики.

Для примера построим таблицу разбора для грамматики со следующими продукциями и их множествами направляющих символов:

$S \to AbB$ $\quad \{a, b, c, e\}$,
$S \to d$ $\quad \{d\}$,
$A \to aAb$ $\quad \{a\}$,
$A \to edAb$ $\quad \{e\}$,
$A \to B$ $\quad \{b, c\}$,
$B \to cSd$ $\quad \{c\}$,
$B \to \varepsilon$ $\quad \{b, d, \perp\}$.

Сначала представим грамматику в виде схемы, в которой все символы помечены номерами строк таблицы разбора:

$$\begin{array}{l} {\scriptstyle 1\quad\ 3\ 4\ 5} \\ S \to Ab\,B, \\ {\scriptstyle 2\quad\ 6} \\ S \to d, \\ {\scriptstyle 7\quad\ 10\,11\,12} \\ A \to a\,A\,b, \\ {\scriptstyle 8\quad\ 13\,14\,15\,16} \\ A \to e\,d\,A\,b, \\ {\scriptstyle 9\quad\ 17} \\ A \to B, \\ {\scriptstyle 18\quad\ 20\,21\,22} \\ B \to c\,S\,d, \\ {\scriptstyle 19\quad\ 23} \\ B \to \varepsilon. \end{array}$$

На основании этой схемы легко строится таблица разбора в соответствии с рассмотренными выше правилами заполнения ее полей (табл. 1).

Можно сократить число элементов в таблице разбора. Заметим, что имеет место взаимно однозначное соответствие между нулевым значением поля *jump* и значением **true** поля *return*. Следовательно, вместо проверки поля *return* для определения, следует ли брать номер следующей строки из стека, достаточно проверить поле *jump* на равенство нулю. Это позволяет исключить из таблицы разбора поле (столбец) *return*.

Рассмотрим работу анализатора на примере разбора строки *edcddbb*⊥, которая выводится в соответствии со следующей левосторонней схемой

$S\perp \Rightarrow abB\perp \Rightarrow edAbbB\perp \Rightarrow edBbbB\perp \Rightarrow edcSdbbB\perp \Rightarrow$
$\Rightarrow edcddbbB\perp \Rightarrow edcddbb\perp$.

Таблица 1

	terminals	*jump*	*accept*	*stack*	*return*	*error*
1	{*a*, *b*, *c*, *e*}	3	**false**	**false**	**false**	**false**
2	{*d*}	6	**false**	**false**	**false**	**true**
3	{*a*, *b*, *c*, *e*}	7	**false**	**true**	**false**	**true**
4	{*b*}	5	**true**	**false**	**false**	**true**
5	{*b*, *c*, *d*, ⊥}	18	**false**	**false**	**false**	**true**
6	{*d*}	0	**true**	**false**	**true**	**true**
7	{*a*}	10	**false**	**false**	**false**	**false**
8	{*e*}	13	**false**	**false**	**false**	**false**
9	{*b*, *c*}	17	**false**	**false**	**false**	**true**
10	{*a*}	11	**true**	**false**	**false**	**true**
11	{*a*, *b*, *c*, *e*}	7	**false**	**true**	**false**	**true**
12	{*b*}	0	**true**	**false**	**true**	**true**
13	{*e*}	14	**true**	**false**	**false**	**true**
14	{*d*}	15	**true**	**false**	**false**	**true**
15	{*a*, *b*, *c*, *e*}	7	**false**	**true**	**false**	**true**
16	{*b*}	0	**true**	**false**	**true**	**true**
17	{*b*, *c*, *d*, ⊥}	18	**false**	**false**	**false**	**true**
18	{*c*}	20	**false**	**false**	**false**	**false**
19	{*b*, *d*, ⊥}	23	**false**	**false**	**false**	**true**
20	{*c*}	21	**true**	**false**	**false**	**true**
21	{*a*, *b*, *c*, *d*, *e*}	1	**false**	**true**	**false**	**true**
22	{*d*}	0	**true**	**false**	**true**	**true**
23	{*b*, *d*, ⊥}	0	**false**	**false**	**true**	**true**

Процесс разбора показан в табл 2. Содержимое стека представляется строкой, в которой самый левый символ находится в вершине стека, символ # показывает дно стека, столбец *i* есть номер применяемой строки таблицы разбора.

В процессе разбора некоторые терминалы проверяются несколько раз. Такой неоднократной проверки можно избежать, если отложить обнаружение некоторых синтаксических ошибок на более поздний этап (поздний в смысле шагов разбора, но не считанного текста). Тогда можно сократить число строк в таблице разбора.

Таблица 2

Вводимая строка	*I*	Содержимое стека	Комментарии
edcddbb⊥	1	0#	проверка '*e*', перейти к строке 3
edcddbb⊥	3	4,0#	проверка '*e*', в стек поместить 3 + 1 = 4, перейти к строке 7
edcddbb⊥	7	4,0#	'*e*' не совпадает с '*a*', *error* = **false**, перейти к строке 7 + 1 = 8
edcddbb⊥	8	4,0#	проверка '*e*', перейти к строке 13
edcddbb⊥	13	4,0#	'*e*' принимается, перейти к строке 14
dcddbb⊥	14	4,0#	'*d*' принимается, перейти к строке 15
cddbb⊥	15	16,4,0#	проверка '*c*', в стек поместить 15 + 1 = 16, перейти к строке 7
cddbb⊥	7	16,4,0#	'*c*' не совпадает с '*a*', *error* = **false**, перейти к строке 7 + 1 = 8
cddbb⊥	8	16,4,0#	'*c*' не совпадает с '*e*', *error* = **false**, перейти к строке 8 + 1 = 9
cddbb⊥	9	16,4,0#	проверка '*c*', перейти к строке 17
cddbb⊥	17	16,4,0#	проверка '*c*', перейти к строке 18
cddbb⊥	18	16,4,0#	проверка '*c*', перейти к строке 20
cddbb⊥	20	16,4,0#	'*c*' принимается, перейти к строке 21
ddbb⊥	21	22,16,4,0#	проверка '*d*', в стек поместить 21 + 1 = 22, перейти к строке 1
ddbb⊥	1	22,16,4,0#	'*d*' ∉ {*a*, *b*, *c*, *e*}, *error* = **false**, перейти к строке 1 + 1 = 2
ddbb⊥	2	22,16,4,0#	проверка '*d*', перейти к строке 6
ddbb⊥	6	22,16,4,0#	'*d*' принимается, взять из стека 22, перейти к строке 22
dbb⊥	22	16,4,0#	'*d*' принимается, взять из стека 16, перейти к строке 16
bb⊥	16	4,0#	'*b*' принимается, взять из стека 4, перейти к строке 4
b⊥	4	0#	'*b*' принимается, перейти к строке 5
⊥	5	0#	проверка '⊥', перейти к строке 18
⊥	18	0#	'⊥' не совпадает с '*c*', *error* = **false**, перейти к строке 18 + 1 = 19
⊥	19	0#	проверка '⊥', перейти к строке 23
⊥	23	0#	проверка '⊥', взять из стека 0, перейти к строке 0
⊥	0	#	стек пуст, разбор успешно завершен

С целью экономии памяти можно упаковывать элементы таблицы разбора в машинные слова, но это, очевидно, замедлит работу синтаксического анализатора.

Можно написать программу для получения соответствующей таблицы разбора. Модуль синтаксического анализа можно применять многократно для различных компиляторов, так что при наличии соответствующих программных средств можно получить *LL*(1)-анализатор из *LL*(1)-грамматики с минимальными затратами времени.

LL(1)-метод разбора имеет следующие достоинства:

1) никогда не требуется возврат, поскольку это детерминированный метод;

2) время разбора примерно пропорционально длине программы;

3) имеются хорошие диагностические характеристики и возможность исправления ошибок, т. к. синтаксические ошибки распознаются по первому неприемлемому символу, а в таблице разбора есть список возможных символов продолжения;

4) таблица разбора меньше, чем соответствующие таблицы в других методах разбора.

Список рекомендуемой литературы

1. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты. М.: Вильямс, 2001. 768 с.

2. Ахо А., Ульман Д. Теория синтаксического анализа, перевода и компиляции: В 2 т. М.: Мир, 1978. Т1. 612 с.; Т2. 487 с.

3. Компаниец Р.И. Системное программирование. Основы построения трансляторов. М.: КОРОНАпринт, 2000. 286 с.

4. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. М.: Мир, 1979. 654 с.

5. Рейуорд-Смит В. Теория формальных языков. Вводный курс. М.: Радио и связь, 1988. 128 с.

6. Хантер Р. Проектирование и конструирование компиляторов. М.: Финансы и статистика, 1984. 232 с.

ОГЛАВЛЕНИЕ

Учебное издание

ПАВЛОВ Леонид Александрович

НИСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

Конспект лекций

Редактор Л. Г. Григорьева

Подписано в печать 12.03.2003. Формат 60×84/16. Бумага газетная.
Печать офсетная. Гарнитура Times New Roman. Усл. печ. л. 2,79.
Уч.-изд. л. 3,0. Тираж 300 экз. Заказ № 173

Чувашский государственный университет
Типография университета. 428015 Чебоксары, Московский просп., 15