Е.И.ЧИГАРИНА, М.А. ШАМАШОВ

ТЕОРИЯ КОНЕЧНЫХ АВТОМАТОВ И ФОРМАЛЬНЫХ ЯЗЫКОВ

2007

СГАУ

САМАРА

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА»

Е.И.ЧИГАРИНА, М.А.ШАМАШОВ

ТЕОРИЯ КОНЕЧНЫХ АВТОМАТОВ И ФОРМАЛЬНЫХ ЯЗЫКОВ

Допущено учебно-методическим советом по прикладной математике и информатике УМО по классическому университетскому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по специальности и направлению «Прикладная математика и информатика» и по направлению «Информационные технологии»

САМАРА
Издательство СГАУ
2007

УДК 004.43(075)
ББК 32.973
Ч586

Инновационная образовательная программа "Развитие центра компетенции и подготовка специалистов мирового уровня в области аэрокосмических и геоинформационных технологий"

Рецензенты: д-р техн. наук, проф. А. А. К а л е н т ь е в,
д-р техн. наук, проф. М. А. К о р а б л и н

Чигарина Е.И.

Ч586 **Теория конечных автоматов и формальных языков:** учеб. пособие / *Е.И. Чигарина, М.А. Шамашов.* – Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2007. – 96 с. : ил.

ISBN 978-5-7883-0506-6

В данном учебном пособии изложены основные концепции, методы и алгоритмы теории формальных языков - науки, изучающей математические модели языков и имеющей практическую ориентацию на конструирование программной поддержки интеллектуальных языковых интерфейсов для общения человека с ЭВМ в самых различных областях науки и техники.

Пособие ориентировано на студентов очной и заочной форм обучения, изучающих информатику и компьютерные науки, и специалистов, связанных с задачами проектирования системного и прикладного программного обеспечения автоматизированных систем самой различной ориентации. Рекомендуется в качестве учебного пособия для студентов высших учебных заведений, обучающихся по специальностям: 010500 – Прикладная математика и информатика, 010400 – Информационные технологии по курсам «Языки программирования и методы трансляции», «Теория конечных автоматов и формальных языков».

Утверждено редакционно-издательским советом университета в качестве учебного пособия

ISBN 978-5-7883-0506-6

Оглавление

Введение

Лингвистика – наука о языке. **Математическая лингвистика** – наука, занимающаяся формальными методами построения и изучения языков.
Теория формальных грамматик – раздел математической лингвистики, включающий способы описания формальных грамматик языков, построение методов и алгоритмов анализа принадлежности цепочек языку, а также алгоритмов перевода (трансляции) алгоритмических языков на язык машины.

Импульсом к созданию и совершенствованию этой теории послужило развитие вычислительной техники и, как следствие, необходимость в средствах общения человека с ЭВМ. Во всех применениях ЭВМ должна понимать какой-либо язык, на котором пользователь может сообщить ей алгоритмы решения задач и исходные данные. Каждая ЭВМ имеет собственный язык машинных команд, представляемых в двоичном коде и отражающих отдельные операции процессора. Автоматизация программирования привела к созданию вначале языков ассемблера, а затем и алгоритмических языков высокого уровня, перевод с которых на родной машинный язык был поручен самой ЭВМ. Программы такого перевода называются *трансляторами*.

С проблемами объяснения языка машине сталкиваются многие разработчики программного обеспечения. Человеку свойственно изобретать новые языки. Здесь речь может идти не только о сложных компиляторах для новых алгоритмических языков программирования. Любая автоматизированная система должна понимать некоторый входной язык запросов. Новые информационные технологии предполагают привлечение конечного пользователя (ученого, конструктора, технолога, оператора) - специалиста в конкретной области, а не в области вычислительной техники и технологии программирования, к решению своих задач на ЭВМ. Для качественного решения этой проблемы между пользователем и ЭВМ должен существовать интеллектуальный интерфейс, - пользователь должен ставить задачи и получать результаты их решения в терминах известной ему предметной области. То есть необходима разработка широкого спектра предметно-ориентированных языков. Специалист в области программного обеспечения должен знать, как создаются языки и их программная поддержка.

Чтобы объяснить язык машине, необходимо четко представлять, как он устроен и как мы его понимаем. Задумавшись над этим, мы увидим, что не знаем, как мы понимаем наш родной язык. Процесс этого понимания подсознателен, интуитивен. Но чтобы создать транслятор, необходимо иметь алгоритм перевода текста в те действия, которые следует выполнить, а это, в свою очередь, требует *формализации языка*. Задачи формализации языка и решает математическая лингвистика. Естественные языки слишком сложны, и формализовать их полностью пока не удается. Алгоритмические языки, напротив, уже создаются в расчете на формализацию. Теория формальных языков - это наиболее развитая ветвь математической лингвистики,

являющаяся, по сути, методикой объяснения языка машине. Прежде чем рассматривать определения, модели и методы этой теории, рассмотрим некоторые понятия на примерах из естественных языков.

Язык – это множество предложений (фраз), построенных по определенным правилам.

Грамматика – свод правил, определяющих принадлежность фразы языку.

Любой язык должен удовлетворять свойствам разрешимости и однозначности.

Язык разрешим, если за конечное время можно определить, что фраза или предложение принадлежит языку. **Язык однозначен**, если любая фраза понимается единственным образом.

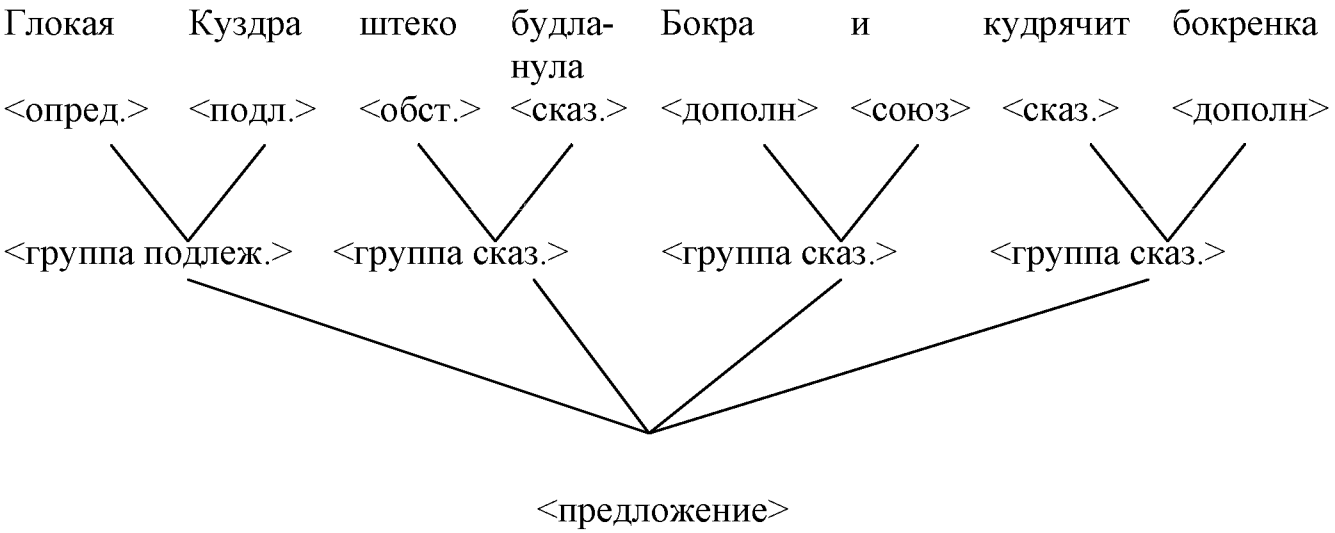
Основными разделами грамматики являются синтаксис и семантика.

Синтаксис – свод правил, определяющих правильность построения предложений языка.

Семантика – свод правил, определяющих семантическую или смысловую правильность предложений языка.

Предложение может быть синтаксически верным и семантически неверным.

Синтаксис обычно упрощается тем, что не все фразы языка обязаны иметь смысл. Зачастую трудно понять смысл футуристов или речь некоторых политиков. В этой связи интересен пример академика Л.В.Щербы: «Глокая куздра штеко будланула бокра и кудрячит бокренка». Это фраза на русском языке, так как её можно разобрать по членам предложения, но смысл её неясен. Синтаксический анализ фразы можно записать в виде дерева грамматического разбора. Узлы дерева, такие как подлежащее, сказуемое, группа подлежащего, предложение соответствуют синтаксическим понятиям, а листья – это слова, из которых строится предложение. Обрубив в дереве часть листьев и ветвей, мы получим сентенциальную форму (выводимую цепочку).



Природу неоднозначности фразы можно объяснить на примере все того же дерева разбора для фразы «Мать любит дочь».

Эта фраза двусмысленна, так как имеет два варианта синтаксического разбора. Синтаксическая неоднозначность напрямую влечет неоднозначность семантическую. Но можно предложить и примеры синтаксически однозначных фраз, которые могут быть не поняты из-за неоднозначного смысла слов. Напомним, что алгоритмический язык должен быть однозначным.

Формальный язык – это математическая абстракция, возникшая как обобщение обычных лингвистических понятий естественных языков. Теория формальных языков изучает в основном синтаксис языков и является фундаментом синтаксически управляемых процессов перевода, к которому можно отнести трансляцию, ассемблирование и компиляцию. Основы этой теории были заложены американским математиком Н. Хомским в конце 50-х-начале 60-х годов и до настоящего времени продолжают развиваться вместе с развитием вычислительной техники. Остановимся на основных элементах этой теории.

Глава 1. Языки и грамматики. Обозначения, определения и классификация

1.1 Понятие грамматики языка. Обозначения

Алфавит – непустое, конечное множество символов, использующихся в языке. **Язык –** множество предложений**. Предложение –** цепочка символов некоторого алфавита.

Всякая конечная последовательность символов алфавита называется ***цепочкой****.* Так *a, b, c, ab, aaca* - цепочки в алфавите $A = \{a,b,c\}$.

Допустим существование ***пустой цепочки*** ε*, не содержащей ни одного символа.*

Важен порядок символов в цепочке. Цепочка *ab* отличается от *ba.*

Длина цепочки $|\alpha|$ *равна количеству символов в цепочке*. Так $|a|=1$, $|abc|=3$, $|\varepsilon|=0$.

Если α *и* β *цепочки, то их* ***конкатенацией*** $\alpha\beta$ *является цепочка, полученная путем дописывания символов цепочки* β *вслед за символами цепочки* α. Например, если $\alpha=ab$, $\beta=bc$, то $\alpha\beta=abbc$ и $\beta\alpha=bcab$. Поскольку ε-цепочка, *не содержащая* символов, то в соответствии с правилами конкатенации для любой цепочки α можно записать $\varepsilon\alpha=\alpha\varepsilon=\alpha$.

Обращением *цепочки* α *(обозначается* α^R*)* *называется цепочка* α, *записанная в обратном порядке*, т.е. если $\alpha = a_1...a_n$, где все a_i - символы, то $\alpha^R = a_n... a_1$. Кроме того, $\varepsilon^R=\varepsilon$.

Языком L *в алфавите* A *называется множество цепочек в* A.

Это определение подходит почти для любого языка. Языки *Паскаль, С, Модула-2, английский* и *русский* охватываются этим понятием.

Рассмотрим простые примеры языков в алфавите A.

Пустое множество $\varnothing$***-*** *это язык. Множество* $\{\varepsilon\}$, *содержащее только пустую цепочку, также является языком.*

Заметим, что $\varnothing$ и $\{\varepsilon\}$ - это два разных языка.

Обозначим через A^* *множество, содержащее все цепочки в алфавите* A, *включая* ε.

Например, если A бинарный алфавит $\{0,1\}$, то

$A^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, ...\}$.

Цепочку, состоящую из i *символов* a *будем обозначать* a^i, то есть $a^0=\varepsilon$, $a^1=a$, $a^2=aa$, $a^3=aaa$ и т.д.

Итак, язык L – это некоторое множество цепочек в некотором алфавите A. Как описать этот язык. Если L состоит из конечного числа цепочек, то самый очевидный способ состоит в составлении списка всех цепочек этого языка. Однако для большинства языков нельзя или нежелательно устанавливать верхнюю границу длины самой длинной цепочки. Следовательно, приходится рассматривать язык, содержащий сколь угодно много цепочек. Очевидно, их нельзя описать перечислением цепочек. Мы хотим, чтобы описание языка было конечным (имел конечный объем), хотя сам язык может быть и бесконечным. Известно несколько методов такого описания. Один из основных методов состоит в использовании ***порождающей системы***, называемой ***грамматикой***.

Грамматика языка – множество объектов: G = <S, V_T, V_N, R>

S – начальный, выделенный символ грамматики (аксиома грамматики);

V_T – множество терминальных символов (терминал – конечный, неделимый символ);

V_N– множество нетерминальных символов (понятий) ;

R – множество правил грамматики.

Грамматику языка обозначим через G, возможно с индексом, или $G(S)$. Язык L, определяемый грамматикой G, обозначим $L(G)$.

Терминальные символы обозначаются маленькими латинскими буквами и являются символами алфавита.

Нетерминальные символы обозначаются заглавными латинскими буквами, либо текстом, заключенными в угловые скобки: <нетерминал>.

Цепочки символов обозначаются буквами греческого алфавита. В обозначение цепочки символов могут входить как терминальные, так и нетерминальные символы.

V*- множество цепочек, построенных на алфавите языка V.

Правила обычно записывают в виде $\alpha \to \beta$ или $\alpha ::= \beta$, что означает α ***порождает*** (***состоит из***) β.

Язык, порождаемый грамматикой, *- это множество цепочек, которые состоят только из терминалов и выводятся, начиная с одного, особо выделенного, нетерминала* ***S****, называемого* ***начальным символом*** *или* ***аксиомой*** *грамматики. Среди множества правил грамматики должно присутствовать хотя бы одно правило* $S \to \beta$.

Обозначение правил в форме записи $\xi ::= \eta$ относится к нотации **Бэкуса – Наура** (бэкусовой нормальной форме - БНФ). В этой нотации используются обозначения:

::= - это есть, | - или,

[ ]- факультативный элемент (необязательная часть), то есть конструкция в скобках может присутствовать или отсутствовать во фразе языка,

{ }- множественный элемент (одно из, элемент выбора), то есть во фразе языка используется один из элементов внутри скобки.

Правило вида $\alpha ::= \beta \mid \beta\gamma$ можно представить $\alpha ::= \beta[\gamma]$.

Бэкусовая нормальная форма (***БНФ***) или ***форма Бэкуса-Наура*** была предложена Д.Бэкусом в 1959 году и впервые применена П.Науром для описания языка Алгол-60. ***Метаязыком*** называется язык, который используется для описания других языков.

Правила вида $\xi \to \eta$ являются порождающими, правила вида $\xi \leftarrow \eta$ - распознающими.

Рассмотрим ряд грамматик и обсудим алгоритм порождения, применяемый для вывода цепочек языка.

Пример 1.1.

<предложение>::=<подлежащее><группа сказуемого>
<подлежащее>::=мать | отец
<группа сказуемого>::=<сказуемое><дополнение>
<сказуемое>::=любит | обожает | боготворит
<дополнение>::=сына | дочь □

Если имеется множество правил, то ими можно воспользоваться для того, чтобы ***вывести*** или ***породить*** цепочку (предложение) по следующей схеме. Начнем с начального символа грамматики - *<предложение>*, найдем правило, в котором *<предложение>* слева от ::= , и подставим вместо *<предложение>* цепочку, которая расположена справа от ::=, то есть

<предложение> $\Rightarrow$ *<подлежащее> <группа сказуемого>*.

Таким образом, мы заменяем синтаксическое понятие на одну из цепочек, из которых оно может состоять. Повторим процесс. Возьмем один из

нетерминалов в цепочке <подлежащее> <группа сказуемого>, например <подлежащее>; найдем правило, где <подлежащее> находится слева от ::=, и заменим <подлежащее> в исходной цепочке на соответствующую цепочку, которая находится справа от ::=. Это дает

<подлежащее> < группа сказуемого > $\Rightarrow$ мать < группа сказуемого > .

Символ "$\Rightarrow$" означает, что один символ слева от $\Rightarrow$ в соответствии с правилом грамматики заменяется цепочкой, находящейся справа от $\Rightarrow$. Полный вывод одного предложения будет таким:

<предложение> $\Rightarrow$ <подлежащее> <группа сказуемого>
$\Rightarrow$ мать < группа сказуемого >
$\Rightarrow$ мать <сказуемое> <дополнение>
$\Rightarrow$ мать любит <дополнение>
$\Rightarrow$ мать любит сына.

Этот вывод предложения запишем сокращенно, используя новый символ $\Rightarrow^{+}$<предложение> $\Rightarrow^{+}$ мать любит сына.

На каждом шаге можно заменить ***любой*** нетерминал. В приведенном выше выводе всегда заменялся самый левый из них.

Вывод, на каждом шаге которого заменяется самый левый нетерминал сентенциальной формы, называется ***левым*** (***левосторонним***) ***выводом***. Существует и часто используется также ***правый*** (***правосторонний***) ***вывод***, который получается, если в сентенциальной форме заменять всегда самый правый нетерминал.

Обратите внимание на то, что предложенная грамматика используется для описания многих предложений. Девять правил грамматики, если считать каждую альтернативу за отдельное правило, а так оно и есть, определяют двенадцать предложений (цепочек) языка:

мать любит сына	*мать обожает сына*	*мать боготворит сына*
мать любит дочь	*мать обожает дочь*	*мать боготворит дочь*
отец любит сына	*отец обожает сына*	*отец боготворит сына*
отец любит дочь	*отец обожает дочь*	*отец боготворит дочь*

Одно из назначений грамматики как раз и состоит в том, чтобы описывать ***все*** цепочки языка с помощью приемлемого числа правил. Это особенно важно, если учесть, что количество предложений в языке, чаще всего, бесконечно.

Рассмотрим еще один пример полезной грамматики

Пример 1.2. Грамматики целого числа без знака содержат следующие 13 правил

(1) <число>::=<чс>	$S \to A$
(2) <чс>::=<цифра><чс>	$A \to AB$
(3) <чс>::=<цифра>	$A \to B$
(4) <цифра>::=0	$B \to 0$
(5) <цифра>::=1	$B \to 1$
(6) <цифра>::=2	$B \to 2$
(7) <цифра>::=3	$B \to 3$
(8) <цифра>::=4	$B \to 4$

(9) <*цифра*>::=*5* $B \to 5$

(10) <*цифра*>::=*6* $B \to 6$

(11) <*цифра*>::=*7* $B \to 7$

(12) <*цифра*>::=*8* $B \to 8$

(13) <*цифра*>::=*9* $B \to 9.$

Заметим, что грамматики *G(<число>)* и *G(S)* определяют один и тот же язык и отличаются только именами нетерминалов и вторым правилом □

А теперь продолжим наши определения.

Пусть ***G*** *- грамматика. Будем говорить, что цепочка* α ***непосредственно порождает*** *цепочку* β*, и обозначим* $\alpha \Rightarrow \beta$*, если для некоторых цепочек* φ *и* ψ *можно написать* $\alpha = \varphi U \psi$, $\beta = \varphi\gamma\psi$, *где* $U ::= \gamma$ *правило грамматики* ***G****. Будем также говорить, что* β ***непосредственно выводима*** *из* α *или что* β ***непосредственно приводится (редуцируется, сворачивается)*** *к* α.

Цепочки φ и ψ, конечно, могут быть пустыми. Следовательно, для любого правила $A \to \alpha$ грамматики G имеет место $A \Rightarrow \alpha$. На рис. 1.1 даны некоторые примеры непосредственных выводов для грамматики *G(<число>)* из примера 1.2 и обозначений предыдущего определения.

Будем говорить, что α ***порождает*** β *или* β ***приводится*** *к* α *и записывать* $\alpha \Rightarrow^{+} \beta$*, если существует последовательность непосредственных выводов* $\alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_n = \beta$, *где n>0. Эта последовательность называется* ***выводом длины n****. Будем писать* $\alpha \Rightarrow^{*} \beta$, *если* $\alpha \Rightarrow^{+} \beta$ *или* $\alpha = \beta$.

α		β	Номера правил	φ	ψ
<число>	⇒	<чс>	1	ε	ε
<чс>	⇒	<цифра> <чс>	2	ε	ε
<цифра> <чс>	⇒	2 <чс>	6	ε	<чс>
2 <чс>	⇒	2 <цифра>	3	2	ε
2 <цифра>	⇒	25	9	2	ε

Рисунок 1.1. Вывод для грамматики целых чисел без знака

Если просмотреть все строки на рисунке 1.1, то мы получим <*число*> $\Rightarrow$ <*чс*> $\Rightarrow$ <*цифра*> <*чс*> $\Rightarrow$ *2* <*чс*> $\Rightarrow$ *2* <*цифра*> $\Rightarrow$ *25.*

Таким образом, <*число*> $\Rightarrow^{+}$ *22* и длина вывода равна *5*. (Если длина вывода известна можно, записывать в явном виде <*число*> $\Rightarrow^{5}$ *22*)

Заметим, что пока в цепочке есть хотя бы один нетерминал, из нее можно вывести новую цепочку, однако если нетерминальные символы отсутствуют, то вывод завершен. Неслучайно "*терминалом*" (*terminal* - заключительный, конечный) называют символ, который не встречается в левой части ни одного из

правил. Исключение составляют нетерминалы-тупики, которые будут рассмотрены позже.

Цепочка $\boldsymbol{\alpha}$ *называется* ***сентенциальной формой****, если* $\boldsymbol{\alpha}$ *выводима из начального символа* $\boldsymbol{S}$*, то есть, если* $\boldsymbol{S \Rightarrow^* \alpha}$.

Цепочка языка - это сентенциальная форма, состоящая только из терминалов.

Язык $\boldsymbol{L(G(S))}$ *- это множество цепочек:* $\boldsymbol{L(G)=\{\alpha \mid S \Rightarrow^* \alpha}$ *и* $\boldsymbol{\alpha \in V_T^*\}}$*, то есть язык - это подмножество множества всех терминальных цепочек* $\boldsymbol{V_T^*}$.

Структура цепочек языка задается грамматикой и, как видно из примера 1.2, *несколько грамматик могут определять один и тот же язык.* Такие грамматики называются ***эквивалентными***.

Пусть $\mathbf{G(S)}$ *- грамматика. И пусть* $\boldsymbol{\omega = \alpha\beta\gamma}$ *- сентенциальная форма. Тогда* $\boldsymbol{\beta}$ *называется* ***фразой*** *сентенциальной формы* $\boldsymbol{\omega}$ *для нетерминального символа* $\boldsymbol{U}$*, если* $\boldsymbol{S \Rightarrow^* \alpha U \gamma}$ *и* $\boldsymbol{U \Rightarrow^+ \beta}$*; и далее,* $\boldsymbol{\beta}$ *называется* ***простой фразой****, если* $\boldsymbol{S \Rightarrow^* \alpha U \gamma}$ *и* $\boldsymbol{U \Rightarrow \gamma}$.

Тот факт, что $U \Rightarrow^+ \beta$, вовсе не означает, что β является фразой сентенциальной формы $\alpha\beta\gamma$, необходимо также иметь $S \Rightarrow^* \alpha U\gamma$. Значит ли в примере 1.2, что *<чс>* является фразой сентенциальной формы *1 <чс>*, если существует правило *<число>::=<чс>* ? Конечно, нет, поскольку невозможен вывод цепочки *1 <число>* из начального символа грамматики *<число>*. Каковы же фразы сентенциальной формы *1 <чс>* ? Имеет место вывод *<число>* $\Rightarrow$ *<чс>* $\Rightarrow$ *<цифра><чс>* $\Rightarrow$ *1<чс>*.

Таким образом,

(1) *<число>* $\Rightarrow^*$ *<чс>* и *<чс>* $\Rightarrow^+$ *1<чс>*

(2) *<число>* $\Rightarrow^*$ *<цифра><чс>* и *<цифра>* $\Rightarrow^+$ *1*.

Следовательно, *1<чс>* и *1* - фразы, простой же фразой будет только *1*.

В дальнейшем мы часто будем говорить о ***самой левой*** *простой фразе сентенциальной формы, которая называется* ***основой***.

Грамматики *G(<число>)* и *G(S)* из примера 1.2 описывают бесконечный язык, то есть язык, состоящий из бесконечного числа цепочек. Это обусловлено тем, что правило *<чс>::=<цифра><чс>* ($A \rightarrow AB$) содержит *<чс>* (A) и в левой, и в правой частях, то есть в некотором смысле символ *<чс>* (A) сам себя определяет.

В общем случае, *если* $\boldsymbol{U \Rightarrow^+ \ldots U \ldots}$*, то говорят, что грамматика рекурсивна по отношению к* $\boldsymbol{U}$*. Если* $\boldsymbol{U \Rightarrow^+ U \ldots}$*, то имеет место* ***левая рекурсия****, а если* $\boldsymbol{U \Rightarrow^+ \ldots U}$*, то -* ***правая рекурсия****. Соответствующие правила называют лево- (право)рекурсивными. Если язык бесконечен, то определяющая его грамматика должна быть рекурсивной.*

1.2. Классификация грамматик по Хомскому

Грамматика G называется ***грамматикой типа 3****,* ***регулярной****,* ***праволинейной*** *или* ***автоматной*** *грамматикой (****А-грамматикой****), если каждое правило из* $\boldsymbol{P}$ *имеет вид:*

$A \to xA$ *(праволинейное правило) или*
$A \to x$ *(заключительное правило), где* $A \in V_N$, $x \in V_T$.

То есть каждое правило такой грамматики содержит единственный нетерминал в левой части, всегда один терминал в правой части, за которым может следовать один нетерминал. Для таких грамматик мы в дальнейшем будем пользоваться термином автоматная (**А-**) грамматика.

Грамматика **G** *называется* ***грамматикой типа 2***, ***бесконтекстной*** *или* ***контекстно - свободной (КС-)*** *грамматикой если ее правила имеют вид:*
$A \to \alpha$, *где* $A \in V_N$, $\alpha \in (V_N \cup V_T)^*$.

То есть в каждом правиле такой грамматики имеет место единственный нетерминал слева и произвольная цепочка из терминалов и нетерминалов справа, возможно и пустая. Замена A на α в сентециальной форме не зависит от того, в каком окружении, в каком контексте находится A.

Грамматика **G** *называется* ***грамматикой типа 1***, ***контекстной, нормальных составляющих (НС-)*** *или* ***контекстно - зависимой (КЗ-)*** *грамматикой, если ее правила имеют вид:*
$\varphi A \psi \to \varphi \alpha \psi$, *где* $A \in V_N$, $\varphi, \psi \in (V_N \cup V_T)^*$ *и* $\alpha \in (V_N \cup V_T)^+$, то есть в каждом правиле нетерминал A в контексте φ и ψ заменяется на непустую цепочку α в том же контексте.

Грамматика G называется грамматикой ***типа 0***, *грамматикой* ***с фразовой структурой*** *или* ***рекурсивно перечислимой*** *грамматикой, если ее правила имеют вид:*
$\alpha \to \beta$, *где на левую и правую части правил не наложено никаких ограничений.* □

Нетрудно заметить, что *грамматики* ***типа i*** *одновременно являются грамматиками* ***типа i-1***. *Исключение составляют* ***укорачивающие*** *КС* ***(УКС)*** *- грамматики, то есть грамматики, содержащие* ***аннулирующие правила*** *типа* $A \to \varepsilon$, *которые не являются КЗ-грамматиками.*

Язык, определяемый грамматикой типа ***i*** *называется* ***языком типа i***.

Из примеров 1.2 и 1.3 следует, что языки чисел и идентификаторов являются КС-языками (тип 2).

Тот факт, что язык определяется грамматикой типа i, еще не означает, что его нельзя породить менее мощной грамматикой типа $i+1$. Например, КС-грамматика с правилами $S \to AS \mid \varepsilon$ и $A \to 0 \mid 1$ порождает язык $\{0,1\}^*$, который конечно же можно определить А-грамматикой $S \to 0S \mid 1S \mid 0 \mid 1$.

Что можно сказать о выделенных классах грамматик и языков в целом? Идеальной с теоретической и практической точек зрения являются А-грамматики и языки. Но их класс слишком узок. Даже язык арифметических выражений не является А языком. Тем не менее, теория автоматных языков повсеместно используется при построении трансляторов. Класс языков типа 0, напротив, очень широк и неразрешим в общем случае. Все остальные языки (тип 1 - тип 3), которые обобщенно называют ***контекстными***, разрешимы. Для них существуют алгоритмы, определяющие принадлежность или непринадлежность цепочек языку за конечное число шагов.

Теорема 1.1. *Любой контекстный язык разрешим.*

Доказательство. Для любого контекстного языка L существует порождающая его грамматика G в ***удлиняющей форме***, у которой для всех правил вывода $\alpha \to \beta$ выполняется условие $|\alpha| < |\beta|$. (Доказательство этого факта будет дано позже). В общем случае для контекстной грамматики без аннулирующих правил выполняется условие $|\alpha| \le |\beta|$. Возьмем анализируемую терминальную цепочку η. Длина исследуемой цепочки должна быть конечной. Тогда если $\eta \in L$, то существует вывод $S = \xi_1 \Rightarrow \xi_2 \Rightarrow \ldots \Rightarrow \xi_{n-1} \Rightarrow \xi_n = \eta$, то есть вывод $S \Rightarrow^n \eta$, где $|\eta| \ge n$, так как каждый шаг вывода удлиняет цепочку не менее, чем на единицу. Число выводов с длиной не более n конечно. Поэтому достаточно проверить выводится ли η одним из них. Если η совпадает с одной из терминальных цепочек, выводимых по заданной грамматике G, не более чем за n шагов, то $\eta \in L(G)$, если нет - $\eta \notin L(G)$. □

Неразрешимость языков типа 0 выводят их из рассмотрения в данном курсе,- нет смысла изучать языки, для которых невозможно определить принадлежность цепочки. Прочие же языки, как следует из теоремы 1.1, могут представлять практический интерес. В дальнейшем мы подробно рассмотрим теорию А - и КС - языков, нашедших широкое распространение при проектировании трансляторов.

1.3. Техника построения КС - и А - грамматик

Приобретение навыков в компактном описании языков в виде грамматик и автоматов является одной из главных задач данной части курса. Задачу можно поставить так: задан вид цепочек языка, требуется описать их с помощью порождающей грамматики.

Контекстно-свободную грамматику строить проще, общий подход ее построения состоит в пошаговой детализации. Разбиваем цепочку на компоненты и вводим нетерминалы, отражающие их суть. Каждую компоненту, полученную на предыдущем шаге, разбиваем на подкомпоненты и так далее, пока не дойдем до терминалов. Для примера рассмотрим такую своеобразную "цепочку", как пассажирский поезд.

Пример 1.3. КС-грамматика пассажирского поезда.

<поезд> → <локомотив><вагоны>

<локомотив> → паровоз | тепловоз | электровоз

<вагоны> → <вагон> | <вагон><вагоны>

<вагон> → купейный | плацкартный | общий | СВ | ресторан

Здесь терминалами мы считаем объекты типа "*электровоз*", "*ресторан*" и т.п., хотя могли спуститься и до стоп-крана, и колесных пар.

Отметим еще раз, что для получения цепочки типа "один или много α", где количество α не ограничено сверху, необходимо использовать рекурсивные правила: *<один или много α> → α | α<один или много α>* или

$<$*один или много* $\alpha>\rightarrow\alpha\,|<$*один или много* $\alpha>\alpha$ или
$<$*один или много* $\alpha>\rightarrow\alpha\,|<$*один или много* $\alpha><$*один или много* $\alpha>$.

Пример 1.4.

КС-грамматика действительного числа, то есть цепочек от полного представления числа, типа *-123.567E-21*, до вырожденных: *0.* или *.1*, т.е. в этом числе обязательна точка и хотя бы одна цифра в целой или дробной части. Терминалами здесь являются знаки - " + " и " - ", цифры от "*0*" до "*9*", десятичная точка - "." и символ "*E*". То есть $\Sigma = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., E\}$. В цепочке $<$*число*$>$ можно выделить следующие основные компоненты: *знак числа*, *целую часть*, *точку* - "." , *дробную часть*, символ "*E*", *знак порядка* и само *значение порядка*. Отдельные компоненты или их группы могут отсутствовать. При таком разложении "." и "*E*" - терминалы, а все остальное нетерминалы, требующие дальнейшей декомпозиции. Так $<$*целая часть*$>$ - это одна цифра или несколько цифр, а $<$*знак*$>$ - "+" или "- ". Нетрудно видеть, что *знаки числа* и *порядка* ничем не отличаются друг от друга и для них можно ограничиться одним понятием - *знак*. С точки зрения синтаксиса нет разницы и между *целой частью*, *дробной частью* и *порядком* и их вполне можно определить одно через другое.

В результате имеем грамматику:

(1) $<$*число*$>\rightarrow<$*знак*$><$*целая часть*$>.<$*дробная часть*$>E<$*знак*$><$*порядок*$>$

(2) $<$*число*$>\rightarrow<$*целая часть*$>.<$*дробная часть*$>E<$*знак*$><$*порядок*$>$

(3) $<$*число*$>\rightarrow<$*знак*$>.<$*дробная часть*$>E<$*знак*$><$*порядок*$>$

(4) $<$*число*$>\rightarrow<$*знак*$><$*целая часть*$>.E<$*знак*$><$*порядок*$>$

(5) $<$*число*$>\rightarrow<$*знак*$><$*целая часть*$>.<$*дробная часть*$>E<$*порядок*$>$

(6) $<$*число*$>\rightarrow<$*знак*$><$*целая часть*$>.<$*дробная часть*$>$

(7) $<$*число*$>\rightarrow.<$*дробная часть*$>E<$*знак*$><$*порядок*$>$

(8) $<$*число*$>\rightarrow<$*целая часть*$>.E<$*знак*$><$*порядок*$>$

(9) $<$*число*$>\rightarrow<$*целая часть*$>.<$*дробная часть*$>E<$*порядок*$>$

(10) $<$*число*$>\rightarrow<$*знак*$><$*целая часть*$>.E<$*порядок*$>$

(11) $<$*число*$>\rightarrow<$*знак*$>.<$*дробная часть*$>E<$*порядок*$>$

(12) $<$*число*$>\rightarrow<$*целая часть*$>.<$*дробная часть*$>$

(13) $<$*число*$>\rightarrow<$*знак*$>.<$*дробная часть*$>$

(14) $<$*число*$>\rightarrow<$*знак*$><$*целая часть*$>.$

(15) $<$*число*$>\rightarrow<$*целая часть*$>.E<$*порядок*$>$

(16) $<$*число*$>\rightarrow .<$*дробная часть*$>E<$*порядок*$>$

(17) $<$*число*$>\rightarrow<$*целая часть*$>.$

(18) $<$*число*$>\rightarrow .<$*дробная часть*$>$

(19) $<$*целая часть*$>\rightarrow<$*цифра*$><$*целая часть*$>$

(20) $<$*целая часть*$>\rightarrow<$*цифра*$>$

(21) $<$*дробная часть*$>\rightarrow<$*целая часть*$>$

(22) $<$*порядок*$>\rightarrow<$*целая часть*$>$

(23) $<$*цифра*$>\rightarrow 0$

(24) $<цифра> \rightarrow 1$
(25) $<цифра> \rightarrow 2$
. . . . . .
(32) $<цифра> \rightarrow 9$
(33) $<знак> \rightarrow +$
(34) $<знак> \rightarrow -$

Используя БНФ (альтернативу и факультатив), эти 34 правила можно переписать в виде:

$<число> \rightarrow [<знак>] <целая\ часть>. [<дробная\ часть>] [E [<знак>] <порядок>] |$
$[<знак>][<целая\ часть>].<дробная\ часть>[E [<знак>] <порядок>]$
$<целая\ часть> \rightarrow <цифра>[<целая\ часть>]$
$<дробная\ часть> \rightarrow <целая\ часть>$
$<порядок> \rightarrow <целая\ часть>$
$<цифра> \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
$<знак> \rightarrow + | -$.

Техника построение А-грамматик определяется видом их правил, где в правой части первым должен стоять терминал. Для формирования таких правил просматриваются исходные цепочки, выписываются терминалы, которые можно встретить на очередном шаге, и вводятся нетерминалы, описывающие остаток цепочки. Созданные нетерминалы детализируются точно также как и исходный, то есть просматривается остаток цепочки и т.д.

Пример 1.5. Автоматная грамматика действительного числа.

Число может начинаться с "+" или "-", и тогда оставшуюся часть числа резонно назвать *числом без знака*. Число также может начаться любой цифрой (*0 - 9*) и остаток цепочки будет тем же *числом без знака*. Наконец, число может начаться десятичной точкой ".", и остаток такой цепочки уже иной. Если число без знака может в качестве продолжения содержать цифры и должно завершиться той же точкой, то после точки идет фрагмент цепочки, который точки уже не содержит. Есть смысл назвать этот фрагмент *дробной частью и порядком*. Рассуждая аналогичным образом, мы получим А-грамматику, где использованы следующие сокращения для имен нетерминалов: *чис - число, чбз - число без знака, дчп - дробная часть и порядок, пор - порядок, пбз - порядок без знака:*

$<ч> \rightarrow +<чбз> | -<чбз> | 0<чбз> | 1<чбз> | ... | 9<чбз> | .<дчп>$
$<чбз> \rightarrow 0<чбз> | 1<чбз> | ... | 9<чбз> | .<дчп> | .$
$<дчп> \rightarrow 0<дчп> | 1<дчп> | ... | 9<дчп> | 0 | 1 | ... | 9 | E<пор>$
$<пор> \rightarrow +<пбз> | -<пбз> | 0<пбз> | 1<пбз> | ... | 9<пбз> | 0 | 1 | ... | 9$
$<пбз> \rightarrow 0<пбз> | 1<пбз> | ... | 9<пбз> | 0 | 1 | ... | 9$.

Данная грамматика допускает порождение совсем уж вырожденных цепочек, типа "-." или ".*E1*", т.е. цепочек без целой и дробной частей. На данном этапе проигнорируем этот факт, чтобы не усложнять грамматику.

1.4. Синтаксические деревья вывода в КС-грамматиках. Представление А-грамматик в виде графа состояний. Неоднозначность грамматик

Рассмотрим КС-грамматику с правилами вывода

$$\begin{array}{l} B \to B+B \mid B*B \mid V \mid C \\ V \to a \mid b \mid c \mid \ldots x \mid y \mid z \\ C \to 0 \mid 1 \mid 2 \mid \ldots 8 \mid 9 \end{array} \qquad (\mathbf{1.1.1})$$

и вывод цепочки $b+a*9$

$$B \Rightarrow B+B \Rightarrow B+B*B \Rightarrow V+B*B \Rightarrow V+V*B \Rightarrow V+V*C \Rightarrow b+V*C \Rightarrow b+a*C \Rightarrow b+a*9 \ (\mathbf{1.1.2})$$

Такая запись не очень удобна, так как по ней трудно определить в какой части сентенциальной формы проводилась замена и какой нетерминал породил тот или иной символ. Более наглядна запись в виде ***дерева вывода*** или ***синтаксического дерева***, представленного на рисунке 1.2 (а).

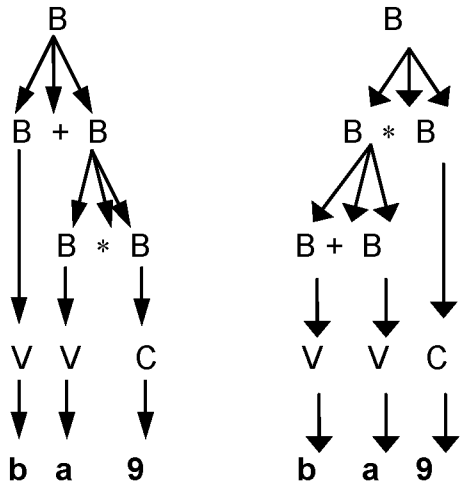
Для того чтобы понять, что выведено, применяем левый обход дерева. Идем от корня по крайней левой ветви, дойдя до терминала (конца ветви), выписываем его, возвращаемся до ближайшего разветвления и идем по самой левой из тех, которые еще не пройдены. Если все ветви данного узла уже исчерпаны, возвращаемся к предыдущему разветвлению, если оно есть. Продолжая таким образом, получим в результате $b+a*9$. Кроме вывода (1.1.2) по данному дереву можно получить целую серию выводов, например,

$$B \Rightarrow B+B \Rightarrow B+B*B \Rightarrow B+B*C \Rightarrow V+V*9 \Rightarrow B+V*9 \Rightarrow B+a*C \Rightarrow V+a*C \Rightarrow b+a*9 (\mathbf{1.1.3})$$

Заметим, что эти выводы отличаются лишь ***порядком*** применения правил и что синтаксическое дерево и грамматика не определяют точный порядок вывода. На каждом шаге вывода имеется некоторый произвол в выборе заменяемого нетерминала. На данном этапе эти различия порядка для нас несущественны и мы считаем выводы эквивалентными, если им соответствует одно и то же дерево. Более важным здесь является то, что цепочка $b+a*9$ в данной грамматике имеет два дерева вывода (рисунки 1.2 (а) и (б)). Сентенциальная форма $B+B*B$ имеет два синтаксических дерева и две основы: $B+B$ и $B*B$. Грамматика неоднозначна, и при разборе сентенциальной формы можно выбрать любую из основ. Нельзя сказать, что выполняется раньше: умножение или сложение. Из рис. 1.2 (б) следует, что $b+a*9$ имеет два подвыражения $b+a$ и 9, хотя по смыслу необходимо иметь подвыражения b и $a*9$.

Цепочка, порождаемая грамматикой, ***неоднозначна****, если для ее вывода существует более одного синтаксического дерева.* ***Грамматика***

неоднозначна, *если она порождает неоднозначные цепочки, в противном случае она* ***однозначна***.



(а) (б)

Рисунок 1.2. Деревья вывода цепочки $b+a*9$

Здесь речь идет о неоднозначной грамматике, а не языке. Изменяя неоднозначную грамматику, можно получить однозначную грамматику для того же самого языка. Ниже приведена однозначная грамматика арифметических выражений

$$\begin{aligned} &\langle врж\rangle \to \langle терм\rangle \mid +\langle терм\rangle \mid -\langle терм\rangle \mid \langle врж\rangle + \langle терм\rangle \mid \langle врж\rangle - \langle терм\rangle \\ &\langle терм\rangle \to \langle множ\rangle \mid \langle терм\rangle * \langle множ\rangle \mid \langle терм\rangle / \langle множ\rangle \quad \textbf{(1.1.4)} \\ &\langle множ\rangle \to (\langle врж\rangle) \mid i \mid k \end{aligned}$$

В этой грамматике i - любой идентификатор (имя переменной), а k - любая константа. Единственное дерево вывода для выражения $i+i*k$ представлено на рисунке 1.3. (а). В соответствии с предложенной грамматикой, эта, да и все остальные цепочки, порождаемые грамматикой (1.1.4) однозначны.

Определим теперь, что в выражении $i+i*k$ должно выполняться раньше: сложение или умножение. Операндами для $+$, согласно дереву, является $\langle врж\rangle$, из которого выводится i, и $\langle терм\rangle$, порождающий $i*k$. Это означает, что умножение должно выполняться первым и образовать $\langle терм\rangle$ для сложения; следовательно, умножение предшествует сложению. Сделать наоборот можно используя только скобки, как показано на рис. 1.3 (б). Грамматику арифметических выражений (1.1.4) следует предпочесть грамматике (1.1.1) ввиду ее однозначности и учета приоритета операций.

Наглядным способом представления А-грамматики является ***граф состояний*** (***переходов***). Вершины этого графа соответствуют нетерминалам, и из вершины A в вершину B проводится дуга, помеченная терминалом a, если в грамматике существует правило $A \to aB$.

Если в грамматике присутствует заключительное правило $A \to b$, а само это правило будем записывать $A \to bF$, получим тем самым ***модифицированную А-грамматику***.

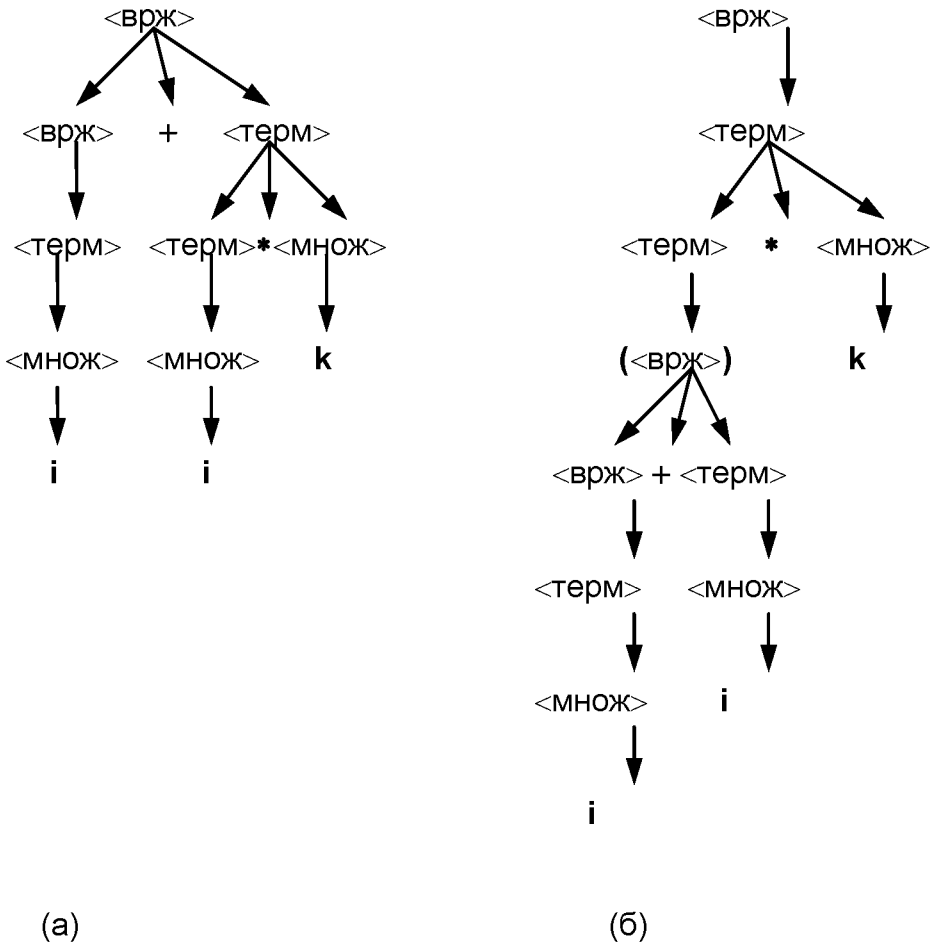


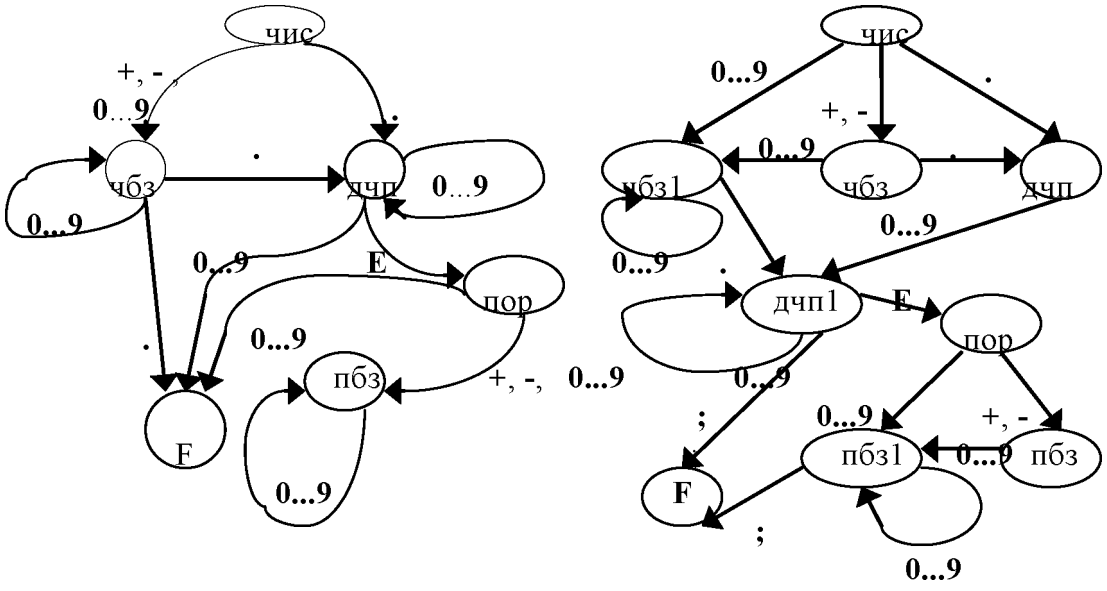
Рисунок 1.3. Деревья вывода выражения $i+i*k$

Заметим, что каждому выводу в А-грамматике будет соответствовать путь по графу состояний из начальной вершины S в вершину F. Граф А-грамматики действительного числа из примера 1.5 представлен на рисунке 1.4 (а).

А-грамматика числа, рассмотренная в примере 1.5, является ***недетерминированной***, то есть в ней существует такой нетерминал A и терминал a, для которых существует несколько нетерминалов B, таких что выполняются правила $A \to aB$. Например, модифицированная А-грамматика числа, что отчетливо видно на рис. 4.1 (а), содержит правила $<чбз> \to .<дчп>$ и $<чбз> \to .F$ или $<дчп> \to 0<дчп>$ и $<дчп> \to 0F$. Это нежелательно с точки зрения построения программ синтаксического анализа.

А-грамматика называется ***детерминированной****, если для любого нетерминала* $\boldsymbol{A}$ $(\boldsymbol{A \neq F})$ *и любого терминала* $\boldsymbol{a}$ *существует не более одного нетерминала* $\boldsymbol{B}$*, для которого выполняется правило* $\boldsymbol{A \to aB}$.

А-грамматика называется ***вполне детерминированной****, если для любого нетерминала* ***A*** *(****A*** *≠* ***F****) и любого терминала* ***a*** *существует единственный нетерминал* ***B****, для которого выполняется правило* ***A*** *→* ***aB****.*



(а) (б)

Рисунок 1.4. Графы состояний А-грамматики действительных чисел без знака

В следующей главе будет показано, что любую недетерминированную А-грамматику можно привести к детерминированной. Пока же, для того чтобы перейти к реализации программ синтаксического анализа А-языков, что является основной частью лабораторных работ на ЭВМ в данной части курса, отметим, что на практике всегда используется специальный символ - ограничитель цепочки. Это либо нулевой байт, либо символ конца файла.

Если, например, считать, что число из примера 1.5 завершается символом “;”, то детерминированная А-грамматика числа строится элементарно. Граф состояний для этого случая представлен на рис. 1.4 (б). Нетерминалы *<чбз1 >*, *<дчп1 >*, *<пбз1 >* добавлены здесь для того, чтобы исключить возможность формирования числа без целой и дробной части и без числа порядка после символа ”*E*”. Вполне детерминированная форма должна включать, кроме того, “ошибочный” нетерминал *< Ош >* и правила типа *<чбз >* → +*<Ош >* или *<дчп >* → .*< Ош >* и т.п. То есть для тех $A \in V_N$ $(A \neq F)$ и $a \in V_T$, для которых в модифицированной детерминированной А-грамматике нет правил $A \rightarrow aB$, необходимо для формирования вполне детерминированной формы добавить правила $A \rightarrow aE$, где $E \equiv$ *<Ош>* - “ошибочная” вершина.

1.5. Синтаксический анализ А-языков

Схема алгоритма и программа, анализирующая принадлежность цепочки заданному А-языку, строиться тривиально по графу состояний А-грамматики. Каждой вершине графа, кроме заключительной и “ошибочной”, соответствует блок выбора очередного символа анализируемой цепочки. Каждому ребру, точнее пометке ребра,- блок анализа символа с последующим переходом к тому или иному блоку выбора. Каждому пути по графу соответствует путь по схеме и программе. Процесс построения анализатора рассмотрим на примере анализатора чисел с фиксированной точкой.

Числа с фиксированной точкой имеют целую и дробную часть и могут описываться, например, следующей автоматной грамматикой:

$S \to +N \mid -N \mid 0F \mid 1C \mid \ldots \mid 9C \mid 0T$

$T \to .D$

$N \to 1C \mid \ldots \mid 9C \mid 0T$

$C \to 0F \mid \ldots \mid 9F \mid 0C \mid \ldots \mid 9C \mid .D$

$D \to 0D \mid \ldots \mid 9D \mid 0F \mid \ldots \mid 9F$.

Для приведения грамматики к вполне детерминированной форме предварительно выполняется ее модификация: вводится предконечное состояние F’, символ конца цепочки - $\perp$, затем все правила вида: $A \to a$ заменяются на $A \to aF'$ и добавляется правило $F' \to \perp F$. Для обозначения ошибочного состояния вводится нетерминал - E.

Теорема: Для любой А-грамматики существует эквивалентная ей А-грамматика во вполне детерминированной форме.

Доказательство: Доказательство включает две части: доказательство существования такой грамматики (оно конструктивное, то есть предлагается алгоритм построения для произвольной автоматной грамматики, автоматной грамматики во вполне детерминированной форме) и доказательство того факта, что построенная грамматика эквивалентна исходной. В этом пункте докажем первую часть теоремы.

Пусть имеется А-грамматика $G=<S,V_T,V_N,R>$. Эта грамматика приводится к модифицированной форме. В результате получим: $V_T=\{\perp,\ldots\}$; $V_N=\{S,F',F,\ldots\}$ А-грамматика во вполне детерминированной форме из модифицированной строится следующим образом: $1.G=<V_T,V_N,<S>,R>$, $<S>=S,<F>=F$, $<F'>=F'$

Если в исходной грамматике имеется правило вида: $A \to aB$, то в новой, построенной грамматике будет правило: $<A> \to a<B>$.

Если в исходной грамматике имеются правила вида: $A \to aB_1 \mid \ldots \mid aB_n$, то в результирующей грамматике будет правило: $<A> \to a<B_1 \ldots B_n>$

Для появившихся новых нетерминалов добавляются правила вида: $<B_1 \ldots B_n> \to c<C_1 \ldots C_n>$, если в исходной грамматике имеются правила следующего вида: $B_1 \to cC_1, \ldots . B_1 \to cCn \ldots B_n \to cC1, \ldots . B_n \to cC_n$

При этом в новый нетерминал нетерминалы C_i включаются один раз. В результате такого построения для любого терминала **a** существует единственное правило вида: $<\ldots A\ldots> \to a<\ldots B\ldots>$. □

Используя описанный алгоритм, приведем к вполне детерминированной форме А-грамматику чисел с фиксированной точкой.

<S>→+<N>|-<N>| 0<FT>|1<C>|…|9<C>

<T>→.<D>

<N>→1<C>|…|9<C>|0<T>

<C>→0<FC>.|…|9<FC>|.<D>

<D>→0<FD>|…|9<FD>

<F'>→⊥<F>

<F'T>→⊥<F>|.<D>

<F'C>→⊥<F>|0<F'C>|…|9<F'C>|.<D>

<F'D>→⊥<F>|0<F'D>|…|9<F'D>. Переобозначим нетерминалы следующим образом: <S> - S, <N> - A, <F"T> - B, <C> - C, <D> - D, <T> - G, <F'C> - H, <F'D> - I. Получим: S→+A|-A|0B|1C|..|9C

A→1C|..|9C|0G

C→0H|..|9H|.D

G→.D

D→0I|…|9I

B→.D|;F

H→0H|…|9H|.D

I→0I|…|9I|;F.

Граф для полученной грамматики имеет вид:

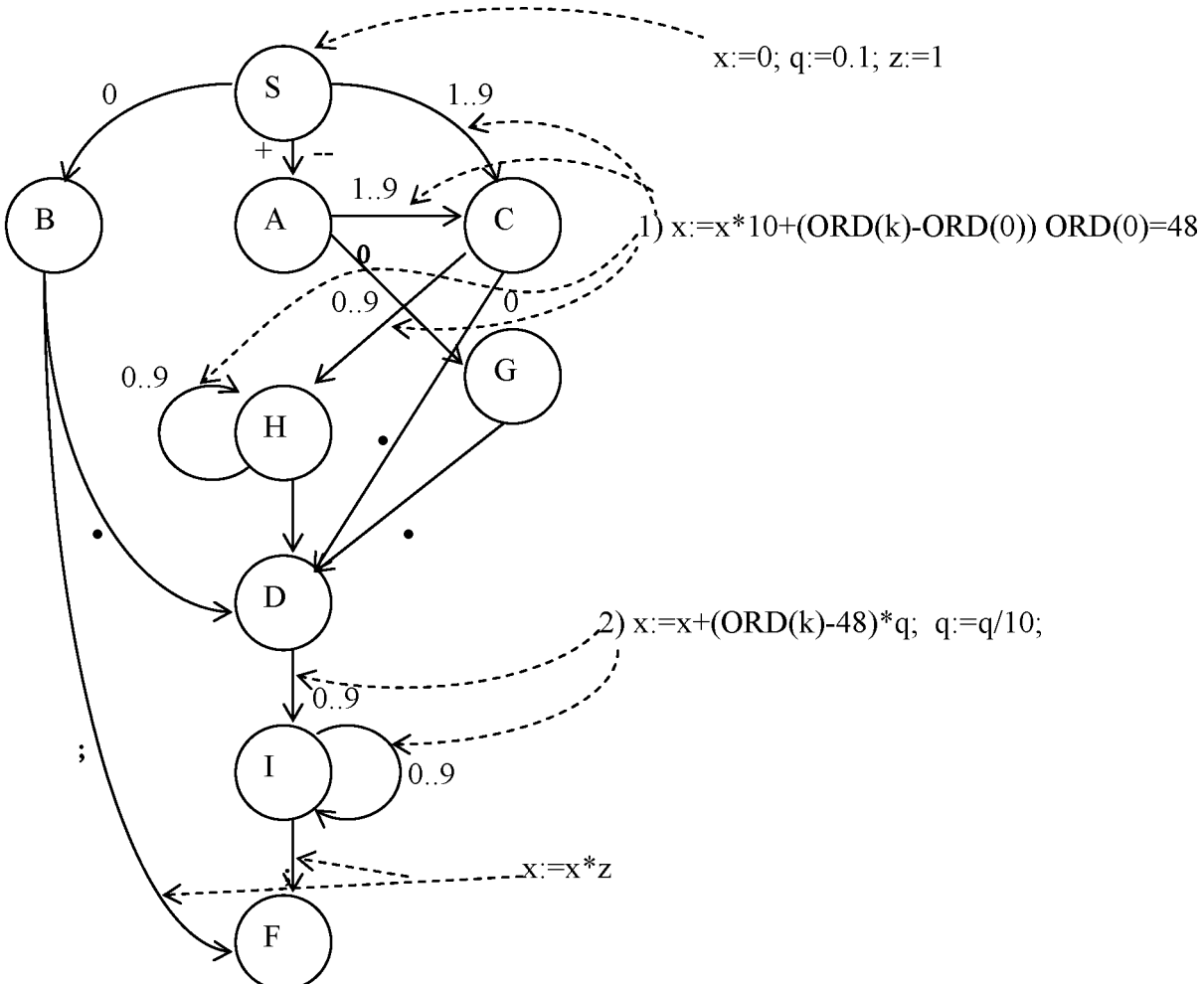


Рисунок 1.5 – Граф состояний А-грамматики чисел с фиксированной точкой

Анализатор - это программа, позволяющая определить принадлежность цепочки языку, порождаемому некоторой грамматикой. Ниже приведена программа, анализирующая правильность написания чисел с фиксированной точкой (автомат Глини) на языке Turbo Pascal.

```
Type Tsost: (S,A,B,C,D,G,H,I,F,E);
Var   Sost : Tsost;  (*Текущее состояние*)
      St : String[255];(*Входная строка-цепочка символов языка*)
      j  : integer;(*Тек. номер позиции в строке*)
      k  : char;(*Тек. символ строки*)
     x,z,q  : real;(*x- число,z – знак, q- значение порядка дробной части числа*)
Begin
      J:=1; sost:=S; read(st);
      X:=0; z:=+1.; q:=0.1;
      While((sost<>F) and(sost<>E)and(j<>length(st))) do
      Begin
                  k:=st[j};
                 Inc(j);
                 Case sost of
                       S: case k of
                         '-': begin
                                    sost:=A;
                                    z:=-1;
                              end;
                         '+': begin
                                    sost:=A;
                              end;
                          '0': begin
                                    sost:=B;
                               end;
                         '1'..'9': begin
                                    sost:=C;
                                    x:=x*10.+(ORD(k)-48)
                                    {48=ORD(0)}
                                end;
                      else
                          begin  writeln('Ошибка-ожидается знак или
цифра');

                                    sost:=E;
                              end;
                      end;{case}
```

```
A: case k of
        '0' : begin
                  sost:=G;
              end;
        '1'..'9' : begin
                       sost:=C;
                       x:=x*10.0+(ORD(k)-48);
                   end;
    else begin writeln('Ошибка в целой части числа');
               sost:=E;
         end:
    end;{case}
 B:case k of
        '.' : begin
                  sost:=D;
              end;
        ';' : sost:=F;
     else(*сообщение об ошибке и sost:=E*)
    end;{case}
C: case k of
        '0'..'9': begin
                       sost:=H;
                       x:=x*10.0+(ORD(k)-48);
                  end;
         '.' : begin
                   Sost:=E;
               end;
     else begin writeln('Ошибка! Ожидается точка');
                sost:=E;
          end;
     end;{case}
G: if k='.' Then sost:=D
          else begin writeln('Ошибка! Ожидается
точка');
                     sost:=E;
               end;
 H: case k of
          '0'..'9' : begin
                       Sost:=H;
                        x:=x*10.+(ORD(k)-48);
                     end;
          '.' : sost:=D;
     else begin writeln('Ошибка! Ожидается точка или
цифра');
                sost:=E;
```

```
                    end;
                end;{case}
            D: case k of
                    '0'..'9' : begin
                                    sost:=I;
                                    x:=x+(ORD(k)-48)*q; q:=q/10;
                end else begin writeln('Ошибка!');
                                sost:=E;
                    end;
                end;{case}
            I: case k of
                    '0'..'9': begin
                                    sost:=I;
                                    x:=x+(ORD(k)-48)*q: q:=q/10:
                                end;
                    ';' : sost:=F;
                else begin writeln('Ошибка!');
                                sost:=E;
                    end;
                end;{case}
            F: writeln('Число сформировано');
            X:=x*z;
            E : writeln('Обнаружена ошибка');
        end{case по состояниям};
    end {while};
end.
```

Семантикой для данного анализатора является значение числа.

Способы включения семантики:

1) применение функции Val(S:String; Var x; Var Code:Integer) (добавляется в состояние F);

2) последовательное формирование числа при переходе автомата в соответствующие состояния:

x:=0, z:=1, q:=0.1

x:=z*(x*10+(значение текущей цифры числа x))

x:=x+(значение текущей цифры числа x)*q, где x – формируемое число, z-знак числа, q – значение прядка текущей цифры дробной части числа.

Алгоритм построения анализатора:

1) Составляется автоматная грамматика.
2) Если полученная грамматика недетерминированная, то она приводится к вполне детерминированной форме.
3) По полученной грамматике строится граф состояний.
4) В соответствие с графом пишется программа.
5) Осуществляется добавление семантических правил в анализатор.

Упражнения к первой главе

1.1. Дайте определение грамматике. Перечислите классы грамматик по Хомскому и дайте их определения. Как определить, содержит ли язык, порождаемый грамматикой, бесконечное число цепочек?

1.2. Что общего в А- и КС- грамматиках? Какие грамматики являются частным случаем других грамматик?

1.3. Постройте КС- и А- грамматики для цепочек вида $x^n y^m z^k$, где $n,k > 0$ и $m \geq 0$. Покажите деревья вывода цепочк *xxxyyz*, *xxzzz* и *xyz* по полученной КС-грамматике.

1.4. Приведите граф состояний для А-грамматики из упр. 1.3. Введите признак конца цепочки, например символ ";", и преобразуйте граф к детерминированной форме. Напишите программу синтаксического анализа цепочек $x^n y^m z^k$, в качестве семантики подсчитайте количество *x*, *y* и *z*.

1.5. Постройте КС-грамматику, А-грамматику и граф состояний для цепочек состоящих из одного или нескольких, идущих без разделения блоков. Каждый блок начинается с единственной буквы и содержит в качестве продолжения от одной до трех цифр.

1.6. Постройте КС-грамматику для цепочек, инвариантных относительно симметричного поворота, т.е. цепочек, которые слева и справа читаются одинаково. Терминалами считайте буквы русского алфавита. Покажите деревья вывода для цепочек *а*, *бб*, *казак*, *шалаш*, *анна*. Подчеркните тот факт, что других цепочек, типа *дом*, *он* по вашей грамматике получить нельзя.

1.7. Постройте КС-грамматику, детерминированную А-грамматику и граф состояний для языка индексов *ФОРТРАН*а. При построении грамматики терминалами считать *{ a b c ...y z 0 1 2...8 9 ( ) + - * }*. Индексы ФОРТРАНа - это ограниченное арифметическое выражение, заключенное в круглые скобки. Ниже перечислены все возможные варианты этих индексов: *(И) (К) (К*И) (И+К) (И-К) (К*И+К) (К*И-К)*, где *И* - идентификатор, а *К* - целая константа без знака.

1.8. Напишите процедуру анализа идентификатора с предупреждением в случае, когда количество символов в нем больше шести; процедуру анализа целой константы без знака с преобразованием символьного представления в число и предупреждением о переполнении (максимальное значение - *65535*). Используя эти процедуры, напишите по А-грамматике программу анализа индексов *ФОРТРАН*а из упражнения 1.7.

1.9. Постройте КС - грамматики для цепочек ($n > 0, m \geq 0$)

(а) $x^n y^n z^m$,

(б) $x^m y^n z^n$,

(в) $x^n y^m z^n$.

1.10. Постройте КС - грамматики для следующих цепочек в бинарном алфавите (Σ={0,1}, $n, m > 0$):

(а) $0^n 1^m 0^m 1^n$;

(б) $\alpha a \alpha^R$, где α - любая цепочка из нулей и единиц, а α^R - обращение (симметричный поворот) цепочки α;
(в) $0^n 1^{2n+3}$;
(г) всех цепочек, содержащих равное количество нулей и единиц;
(д) всех цепочек, содержащих равное количество нулей и единиц и такие, у которых количество нулей в каждой левой подцепочке не меньше чем единиц;
(е) $1^n 0^m 1^p$, где $n+p>m>0$.

1.11. Опишите языки, порождаемые следующими грамматиками с начальным нетерминалом S:

(а) $S \to 10S0$, $S \to aA$, $A \to bA$, $A \to a$

(б) $S \to SS$, $S \to 1A0$, $A \to 1A0$, $A \to \varepsilon$

(в) $S \to 1A$, $S \to B0$, $A \to 1A$, $A \to C$, $B \to B0$, $B \to C$, $C \to 1C0$, $C \to \varepsilon$

(г) $S \to aSS$, $S \to a$

(д) $S \to bADc$, $D \to Gl$, $A \to aGs$, $G \to \varepsilon$.

1.12. Какие из приведенных ниже цепочек можно вывести в данной грамматике? В каждом случае постройте левый вывод, правый вывод и дерево вывода.

$S \to aAcB$, $S \to BdS$
$B \to aScA$, $B \to cAB$
$A \to BaB$, $A \to aBc$
$A \to a$, $B \to b$

(а) *aacb*
(б) *aaacbdaacb*
(в) *aabcbd*
(г) *acabaaaacbcacb*
(д) *aaaaacbcacccab* .

1.13. (а) Найдите КС-грамматику для логических выражений, составленных из логических переменных, констант, скобок и знаков операций отрицания ($\neg$), дизъюнкции ($\vee$) и конъюнкции ($\wedge$). Приоритеты обычные: $\neg$ выполняется перед $\wedge$, а $\wedge$ - перед $\vee$.

(б) Добавьте к указанному языку первичные логические выражения, каждое из которых представляет собой арифметическое выражение, за которым следует знак отношения ($>$, $\geq$, $=$, $\neq$, $<$, $\leq$) и еще одно арифметическое выражение, и постройте соответствующую грамматику.

1.14. Постройте КС-грамматику оператора *FORMAT* языка *ФОРТРАН*, ограничиваясь форматами *A*, *X*, *I*, *F*. Примеры правильных цепочек:

10 FORMAT(I5)
100 FORMAT (X10,A5,3A4,I6,4(I3,X2,5(F8.3,X4,3A2,I6),X2,2(3A3,I4)),I7)

Глава 2. Распознаватели и автоматы

Второй распространенный метод, обеспечивающий задание языка конечными средствами, состоит в использовании *распознавателей* (*автоматов*). В сущности, распознаватель - это схематизированный алгоритм, определяющий некоторое множество. Он состоит из четырех частей: входной ленты, головки чтения (/записи), управляющего устройства с конечной памятью и внешней (вспомогательной, рабочей) памяти (рис. 2.1).

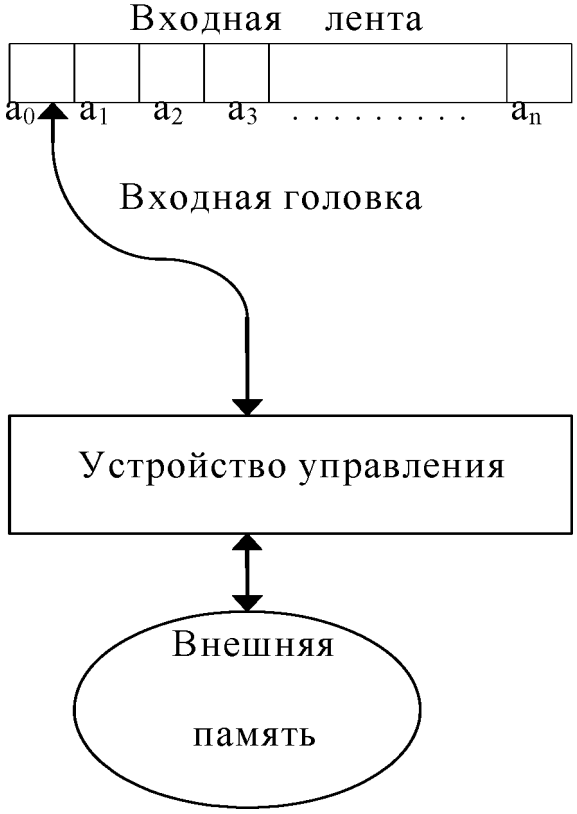


Рисунок 2.1.Схема распознавателя

Входную ленту можно рассматривать как линейную последовательность клеток (ячеек). Причем в каждой ячейке может содержаться только один входной символ из некоторого конечного алфавита Σ. Самую левую и самую правую ячейки могут занимать особые символы - *концевые маркеры*; маркер может стоять только на правом конце ленты; его может не быть совсем.

Входная головка в каждый данный момент читает (обозревает) одну входную ячейку. За один шаг работы распознавателя входная головка может сдвинуться на ячейку вправо R, остаться неподвижной N или сдвинуться влево L. Распознаватель, который никогда не передвигает свою входную головку влево, называется *односторонним*. Обычно предполагается, что входная головка *только читает*, т.е. в ходе работы распознавателя символы на входной ленте не меняются. Но можно рассматривать и такие распознаватели - *преобразователи*,

входная головка которых не только читает, но и пишет, например, *машина Тьюринга общего вида*.

Внешняя память распознавателя - это хранилище информации (данных) некоторого типа. *Алфавит памяти* Z конечен, и информация в памяти образована только из символов этого алфавита. Предполагается, что в любой момент времени можно конечными средствами описать содержимое и структуру памяти, хотя с течением времени и работы распознавателя, объем памяти может становиться сколь угодно большим.

Поведение вспомогательной памяти для заданного класса распознавателей можно охарактеризовать с помощью двух функций: *функции доступа к памяти* (ФДП) и *функции преобразования памяти* (ФПП).

ФДП - это отображение множества *состояний* (*конфигураций*) памяти в конечное множество *информационных символов*, которое может совпадать с алфавитом памяти. Например, единственная информация, доступная в каждый данный момент в магазине (стеке), - верхний символ магазина. Таким образом, функция доступа к магазинной памяти f - это такое отображение Z^* в Z, где Z - алфавит памяти, что

$f(z_1z_2...z_n) = z_1$,

где $z_i \in Z$ для всех $i = \overline{1,n}$.

ФПП - это отображение, описывающее изменение памяти. Она отображает состояние памяти и *управляющую цепочку* в состояние памяти. Если предполагается, что операция над магазинной памятью заменяет верхний символ конечной цепочкой символов памяти, то соответствующую ФПП можно определить как такое отображение g: $Z^+ \times Z^* \to Z^*$, что $g(z_1z_2...z_n,x_1x_2...x_m) = x_1x_2...x_mz_2...z_n$, где z_i, $x_j \in Z$ для всех $i = \overline{1,n}$ и $j = \overline{1,m}$.

Если верхний символ магазина z_1 заменяется пустой цепочкой, то z_2 станет верхним символом магазина.

Вообще именно тип внешней памяти определяет название распознавателя. Например, распознаватель, память которого организована как магазин, называется распознавателем с магазинной памятью. (Обычно его называют автоматом с магазинной памятью или МП-автоматом.)

Ядром же распознавателя является *управляющее устройство* с конечной памятью, под которым можно понимать программу, управляющую поведением распознавателя. Управляющее устройство представляет собой *множество состояний* Q вместе с *отображением* δ, которое описывает, как меняются состояния в соответствии с текущим входным символом (т.е. находящимся под входной головкой) и текущей информацией, извлеченной из внешней памяти. Управляющее устройство определяет также, в каком направлении сдвинуть входную головку, и какую информацию поместить в память.

Распознаватель работает, проделывая некоторую последовательность *шагов* или *тактов*. В начале такта читается текущий входной символ и с помощью функции доступа исследуется память. Входной символ и информация из памяти вместе с текущим состоянием управляющего устройства определяют,

каким должен быть следующий такт. Собственно такт состоит из следующих моментов:
(1) входная головка сдвигается на одну ячейку вправо, влево или сохраняет исходное положение;
(2) в память помещается некоторая информация;
(3) изменяется состояние управляющего устройства.

Поведение распознавателя удобно описывать в терминах *конфигурации* - мгновенного снимка распознавателя, на котором изображены:
(1) состояние устройства;
(2) содержимое входной ленты вместе с положением входной головки;
(3) содержимое памяти.

Управляющее устройство распознавателя может быть *недетерминированным*, т.е. для каждой конфигурации существует конечное множество всевозможных следующих шагов, любой из которых распознаватель может сделать, исходя из этой конфигурации. Устройство может быть *детерминированным*, если для каждой конфигурации существует не более одного следующего шага. Недетерминированный распознаватель - удобная математическая абстракция, но ее трудно моделировать на практике.

Конфигурация называется *начальной*, если управляющее устройство находится в заданном *начальном состоянии* q_0, входная головка обозревает самый левый входной символ на ленте, а память имеет заранее установленное начальное содержимое.

Конфигурация называется *заключительной*, если управляющее устройство находится в одном из состояний заранее выделенного *множества заключительных состояний* $F \in Q$, а головка обозревает правый концевой маркер, или, если маркер отсутствует, сошла с правого конца ленты. Часто требуют, чтобы память в заключительной конфигурации тоже удовлетворяла некоторым условиям.

Говорят, что распознаватель *допускает входную цепочку* α, если начиная с начальной конфигурации, в которой α записана на входной ленте, распознаватель может проделать последовательность шагов, завершающуюся заключительной конфигурацией. Недетерминированный распознаватель может при этом проделать много последовательностей шагов, и если хотя бы одна из них заканчивается заключительной конфигурацией, то исходная входная цепочка будет допущена.

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

Для каждого класса грамматик в иерархии Хомского существует естественный класс распознавателей, определяющий тот же класс языков. Справедливы следующие утверждения:

Язык L автоматный тогда и только тогда, когда он определяется односторонним детерминированным конечным автоматом.

Язык L контекстно - свободный тогда и только тогда, когда он определяется односторонним недетерминированным МП - автоматом.

Язык L контекстно-зависимый тогда и только тогда, когда он определяется двухсторонним недетерминированным линейно-ограниченным (ЛО-) автоматом.

Язык L рекурсивно перечислимый (без ограничений) тогда и только тогда, когда он определяется машиной Тьюринга общего вида.

Глава 3. Автоматные грамматики и конечные автоматы

3.1. Автоматные грамматики и конечные автоматы

Автоматную грамматику можно представить в виде таблицы, где строки соответствуют терминалам, а столбцы нетерминалам. На пересечении строки a и столбца A записываются нетерминалы B, которые входят в правила $A \to aB$. На рис. 3.1 представлена таблица переходов, построенная для А-грамматики из примера 1.9. Приведенная таблица представляет собой не что иное, как *функциональную таблицу* или *таблицу переходов* конечного автомата.

Конечный автомат - это частный случай одностороннего распознавателя, где головка движется всегда вправо на каждом такте работы и отсутствует внешняя память.

Конечный автомат M можно определить пятеркой множеств $M = \{Q, \Sigma, \delta, q_0, F\}$, *где*

Q - *конечное множество состояний автомата* (при построении автоматов оно обычно совпадает с множеством нетерминалов грамматики);

Σ - *конечное множество символов внешнего алфавита* (конечное *множество допустимых входных символов*, совпадающих с множеством терминалов);

δ - *функция переходов, отображающая множество состояний и входных символов в множество состояний* $Q \times \Sigma \to Q$ и представляемая, обычно, в виде таблицы переходов (например, из таблицы с рис. 3.1. следует, что δ (*чис, +*) = *чбз*, а δ (*дчп, Е*) = *пор*);

q_0 - *начальное состояние* (на рис. 3.1 - чис);

F - *множество заключительных состояний* (в теории конечных автоматов заключительные состояния называются также *допускающими*, так как они допускают обрыв (завершение) цепочки или, иными словами, допускают пустую цепочку. На рис. 3.1 допускающие состояния *дчп* и *пбз* отмечены единицами в строке под таблицей, остальные состояния там *отвергающие*, обрыв цепочки в них недопустим, и они отмечены нулем в нижней строке таблицы.

3.2. Эквивалентность недетерминированных и детерминированных А-грамматик

Термин недетерминированный автомат может привести в недоумение, так как в русском и других языках понятие автомата, в сущности, не позволяет применить к нему прилагательное “недетерминированный”. Недетерминированный автомат - это формализм для определения множества цепочек, обобщающий понятие конечного автомата (ни о каких случайностях, вероятностях здесь речи нет).

Σ \ N	чис	чбз	дчп	пор	пбз
+	чбз			пбз	
-	чбз			пбз	
0	чбз	чбз	дчп	пбз	пбз
...	чбз	чбз	дчп	пбз	пбз
9	чбз	чбз	дчп	пбз	пбз
.	дчп	дчп			
E			пор		
	0	0	1	0	1

Рис. 3.1.

Недетерминированный автомат (НДА) - это автомат, где значение функции переходов δ - не отдельное состояние, как в детерминированном автомате (ДА), а множество состояний. В общем случае у НДА может быть также не одно, а множество начальных состояний (если начальное состояние не единственно, то эти состояния при изображении таблицы переходов будут помечаться стрелкой). *НДА* изучается, так как для заданного множества иногда легче найти недетерминированное описание.

Говорят, что НДА допускает входную цепочку, если он позволяет связать одно из его начальных состояний с одним из допускающих.

Так автомат, представленный на рисунке 3.2, допускает цепочку 11, так как $B \xrightarrow{1} C \xrightarrow{1} C$ причем B - начальное, а C - допускающее состояния. Существования одной этой последовательности достаточно, чтобы показать допустимость входной цепочки 11 и существование другой последовательности $B \xrightarrow{1} C \xrightarrow{1} A$,

где A - отвергающее, на это не влияет.

Σ \ Q	A	B	C
0	A, B	B	
1	C	C	A, C
	0	1	1

Рисунок 3.2.Таблица перехода автомата,
допускающего цепочку 11

На рисунке 3.3 представлен еще один НДА с входным алфавитом {*а, л, н, о, с, ь*}, который допускает только две цепочки *лассо* и *лань*.

Σ \ Q	q_0	$Л_1$	$Л_2$	$А_1$	$А_2$	$С_1$	$С_2$	Н	О	Ь
а		$А_1$	$А_2$							
л	$Л_1$,$Л_2$									
н					$Н_1$					
о							О			
с				$С_1$		$С_2$				
ь								Ь		
	0	0	0	0	0	0	0	0	1	1

Рисунок 3.3. Автомат, допускающий две цепочки *лассо* и *лань*

Понятие *НДА* приобретает практическое значение, так как *для каждого НДА существует эквивалентный ему ДА*, то есть *ДА*, допускающий в точности те же цепочки, что и *НДА*.

Основная идея построения *ДА* по *НДА* заключается в том, что после обработки отдельной входной цепочки состояние *ДА* будет представлять собой множество всех состояний *НДА*, которые он может достичь из начальных состояний после применения данной цепочки. Переходы *ДА* можно получить из *НДА*, вычисляя множества состояний, которые могут следовать после данного множества при различных входных символах. Допустимость цепочки определяется исходя из того, является ли последнее детерминированное состояние, которого достиг *ДА*, множеством недетерминированных состояний, включающим хотя бы одно допускающее состояние. Результирующий *ДА* будет иметь конечное множество состояний, так как число подмножеств состояний *НДА* конечно.

В общем случае, если *НДА* содержит *n* состояний, то *ДА* будет содержать 2^n состояний. На практике многие из этих подмножеств представляют собой недостижимые состояния.

В приведенной ниже процедуре перехода от *НДА* к *ДА* переходы строятся только для тех подмножеств, которые действительно необходимы.

Пусть $M_н$ - *НДА*, а $M_д$ - эквивалентный ему *ДА*. Процедура перехода от *НДА* к *ДА* задается пятью шагами.

1. Пометить первый столбец функциональной таблицы для $M_д$ множеством начальных состояний автомата $M_н$. Применить к этому множеству шаг *2*.

2. По данному множеству состояний X, помечающему столбец таблицы переходов $M_д$, для которого переходы еще не вычислены, определить те состояния $M_н$, которые могут быть достигнуты из X с помощью каждого входного символа a, и поместить множества последующих состояний Y_a в соответствующие ячейки таблицы $M_д$. Символически это выражается так: если δ функция *НДА*, то функция *ДА* δ' задается формулой
$\delta'(X,a) = \{y \mid y \in \delta(x,a)$, для некоторого x из $X\}$

3. Для каждого множества Y, порожденного переходами на шаге *2*, определить, имеется ли уже в $M_д$ столбец, помеченный этим множеством. Если нет, то создать новый столбец и пометить его этим множеством Y. Если множество Y уже использовалось как метка, то никаких действий не требуется.

4. Если в таблице $M_д$ есть столбец, для которого переходы еще не вычислены, вернуться назад и применить к этому столбцу шаг *2*. Если все переходы вычислены, перейти к шагу *5*.

5. Пометить столбец как допускающее состояние $M_д$, когда он содержит допускающее состояние $M_н$. В противном случае пометить как отвергающее.

На рисунках 3.4 (а) и 3.5 представлены функциональные таблицы *ДА*, полученные по предложенному алгоритму из *НДА* с рисунков 3.2 и 3.3 соответственно. На рисунке 3.4 (б) та же таблица, что и на 3.4 (а), но с новыми именами состояний. Во всех этих таблицах пустая ячейка - не что иное, как ошибочное состояние.

Q / Σ	{A,B}	{C}	{A,C}
0	{A,B}		{A,B}
1	{C}	{A,C}	{A,C}
	1	1	1

(а)

Q / Σ	A	B	C
0	A		A
1	B	C	C
	1	1	1

(б)

Рисунок 3.4-ДА, соответствующий автомату на рисунке 3.2

Приведенный алгоритм можно использовать и для перехода от недетерминированной А-грамматики к детерминированной.

Пример 3.1. Пусть задана недетерминированная А-грамматика вида:

$S \to aB \mid aC \mid bB \mid bS \mid c$
$B \to cC \mid c$
$C \to a \mid aS \mid c$

Эта грамматика не имеет естественного символа, ограничивающего цепочку, поэтому во всех правилах вида $A \to a$ добавим предзаключительное состояние $F^{/}$ и получим правила $A \to a\,F^{/}$.

Добавим также символ ограничения цепочки, например, $\perp$ и правило $F^{/} \to \perp F$, где F - заключительное состояние. В результате получим модифицированную грамматику

$S \to aB \mid aC \mid bB \mid bS \mid cF^{/}$
$B \to cC \mid cF^{/}$
$C \to aF^{/} \mid aS \mid cF^{/}$
$F^{/} \to \perp F$.

Q / Σ	q_0	{Л$_1$Л$_2$}	{А$_1$А$_2$}	С$_1$	С$_2$	О	Н	Ь
л	{Л$_1$Л$_2$}							
а		{А$_1$А$_2$}						
с			С$_1$	С$_2$				
о					О			
н			Н					
ь							Ь	
	0	0	0	0	0	1	0	1

Рис. 3.5.

Рассматриваем для недетерминированных правил неупорядоченные безповторные достижимые подмножества множества нетерминалов и в результате получим детерминированную грамматику

$S \to a\langle BC\rangle \mid b\langle BS\rangle \mid cF^{/}$
$\langle BC\rangle \to a\langle SF^{/}\rangle \mid c\langle CF^{/}\rangle$
$\langle BS\rangle \to a\langle BC\rangle \mid b\langle BS\rangle \mid c\langle CF^{/}\rangle$
$\langle SF^{/}\rangle \to a\langle BC\rangle \mid b\langle BS\rangle \mid cF^{/} \mid \perp F$
$\langle CF^{/}\rangle \to a\langle SF^{/}\rangle \mid cF^{/} \mid \perp F$
$F^{/} \to \perp F.$

Полученная грамматика порождает те же цепочки, что и исходная. Отметим, что дерево вывода в автоматной грамматике вырождено, и его вполне можно представлять в виде строки. Так дерево вывода цепочки $acac\perp$ в исходной грамматике имеет вид $S\ _a B\ _c C\ _a S\ _c F'\ _\perp F$, а в полученной детерминированной $S\ _a\langle BC\rangle\ _c\langle CF'\rangle\ _a\langle SF'\rangle\ _c F'\ _\perp F.$

Отличие состоит в том, что каждый шаг порождения в детерминированной грамматике однозначен, что и определяет преимущество таких грамматик с точки зрения практического построения анализатора по грамматике. Можно переименовать нетерминалы сформированной детерминированной грамматики и получить грамматику в традиционной форме:

$S \to aA \mid bB \mid cF'$
$A \to aC \mid cD$
$B \to aA \mid bB \mid cD$
$C \to aA \mid bB \mid cF' \mid \perp F$
$D \to aC \mid cF' \mid \perp F$
$F' \to \perp F \square$

Упражнения к третьей главе

3.1. Постройте детерминированные А-грамматики для следующих цепочек, завершающихся символом “;”:
(а) целых чисел без знака и без значащих нулей;
(б) четных целых чисел без знака;
(в) идентификаторов, содержащих четное количество символов;
(г) для цепочек из упражнения 1.5.

3.2. Постройте конечный автомат, который будет распознавать слова *go to* , причем между ними может быть произвольное число пробелов, включая нулевое.

3.3. Постройте конечные распознаватели для описанных ниже множеств цепочек из нулей и единиц:
(а) число единиц четное, а нулей - нечетное;
(б) между вхождениями единиц четное число нулей;
(в) за каждым вхождением пары *11* следует *0*;
(г) каждый третий символ - единица;
(д) имеется по крайней мере одна единица.

3.4. Постройте конечный автомат с входным алфавитом *{0,1}*, который допускает в точности такое множество цепочек:
(а) все входные цепочки;
(б) все входные цепочки, кроме пустой;

(в) ни одной входной цепочки;
(г) входную цепочку *101*;
(д) две входные цепочки: *01* и *0100*;
(е) все входные цепочки, начинающиеся с *0* и заканчивающиеся на *1*;
(ж) все цепочки, не содержащие ни одной единицы;
(з) все цепочки, содержащие в точности три единицы;
(и) все цепочки, в которых перед и после каждой единицы стоит *0*;

3.5. Опишите словами множества цепочек, распознаваемых каждым из следующих автоматов:

\	A	B	C
0	B	B	C
1	A	C	C

0 0 1
(а)

\	A	B	C
0	B	C	C
1	C	B	C

0 1 0
(б)

\	A	B	C	D
0	B	D	C	D
1	C	B	D	D

0 1 1 0
(в)

\	A	B	C	D
0	B	D	C	D
1	A	C	D	D

0 0 1 0
(г)

Рис. 3.6

3.6. Постройте детерминированную А-грамматику по следующей недетерминированной:
$S \to aA|aB|bB|bD$
$A \to aB|aS|bD$
$B \to cS|cB|bD|b$
$D \to dD|dB|bB|bA|a|c$

3.7. Опишите словами множества цепочек, распознаваемых каждым из недетерминированных автоматов, изображенных на рисунке.

\	A	B	C
0	A,B		C
1		B,C	

0 0 1
(а)

\	A	B	C
0	B		
1		C,A	

0 0 1
(б)

\	A	B	C	D
0	B	C,D	B	
1		B		

0 0 0 1
(в)

Рис. 3.7.

3.8. Найдите детерминированный автомат, эквивалентный следующему недетерминированному:

	1	2	3
a	2,3		1
b	2	1	3

0 0 1

Рис. 3.8.

Глава 4. Эквивалентные преобразования контекстно-свободных и автоматных грамматик

4.1. Декомпозиция правил грамматики

Определение: две грамматики эквивалентны, если они порождают один и тот же язык. $G_1 \sim G_2$, если $L(G_1)=L(G_2)$.

Теорема: не существует алгоритма, определяющего эквивалентность или неэквивалентность двух грамматик, тем не менее существуют такие преобразования исходных грамматик, которые приводят к новым грамматикам, не выходящим из класса грамматик, эквивалентных исходным.

Критериями преобразования грамматик, приведения к эквивалентным грамматикам являются следующие: детерминированность; получение более короткого вывода цепочек; исключение тупиковых правил.

В предыдущей главе была сформулирована теорема о существовании для любой А-грамматики эквивалентной ей грамматики во вполне детерминированной форме и доказана первая часть теоремы, подтверждающая существование таких грамматик. Ниже будет доказана вторая часть теоремы, рассматривающая эквивалентность исходной А-грамматики и построенной.

Для доказательства эквивалентности исходной автоматной грамматики и построенной грамматики во вполне детерминированной форме необходимо доказать, что любая цепочка, принадлежащая языку L_1, выводится и в грамматике G_2 и наоборот: $L_1=L_2$, если $L(G_1) \subset L(G_2)$ и $L(G_2) \subset L(G_1)$.

Рассмотрим произвольную цепочку $\varphi=a_1 \ldots a_k \in L(G_1)$, тогда существует вывод этой цепочки вида:

$$S_{a_1} A_{1_{a_2}} A_2 ... a_2 F$$

Согласно построению, если в исходной грамматике существует правило вида $S \rightarrow aA_1$, то в преобразованной грамматике будет правило вида $\langle S \rangle \rightarrow a\langle ..A_1 \ldots \rangle$.

Аналогично рассуждаем для произвольного шага: если $A_i \rightarrow a_{i+1} A_{i+1}$, то $\langle ..A_{i..} \rangle \rightarrow a_{i+1} \langle ..A_{i+1}.. \rangle$

На последнем шаге вывода, имея в исходной грамматике правило вида $A_{k-1} \to a_k F$, в преобразованной грамматике имеем правило $<..A_{k-1}..> \to a_k <..F..>$, то есть существует последовательность шагов вывода:

$<S>_{a1} <..A_1 \ldots>_{..ak} <..F..> \Rightarrow \varphi \in L(G_2)$

В противоположную сторону рассуждения абсолютно аналогичны:

Пусть $\varphi = a_1 \ldots a_k \in L(G_2) \Rightarrow$ существует вывод этой цепочки в языке, порождаемом грамматикой G_2, то есть существует вывод цепочки: $<S>_{a1} <..A_1 \ldots>_{..ak} <..F..>$.

По построению, если существует правило вида $<S> \to a_1 <..A_1 \ldots>$, то в исходной грамматике существует правило вида $S \to a_1 A_1$.

Рассуждая аналогично, имея правило вида $<..A_{k-1}..> \to a_k <..F..>$ в исходной грамматике, имеем правило вывода $A_{k-1} \to a_k F$, то есть $\varphi \in L(G_1)$ и $\Rightarrow L(G_2) \subset L(G_1)$, откуда $L_1 = L_2$. Что и требовалось доказать.

Следующие теоремы позволяют обосновать возможность эквивалентных преобразований, приводящих к грамматикам с большим количеством правил, но вывод в которых короче.

Пусть дана КС-грамматика $G_1 = (V_N, V_T, R, S)$.

Теорема 4.1. *Если в КС-грамматике G_1 существуют правила $Y \to \alpha X \beta$ и $X \to \gamma$, то грамматика $G_2 = (V_N, V_T, R \cup \{Y \to \alpha\gamma\beta\}, S)$ эквивалентна G_1.*

Доказательство. Проверим, что любая цепочка, выводимая в одной грамматике, выводима и в другой.

Пусть $\varphi \in L(G_1)$, тогда дерево вывода φ в G_1 является деревом вывода φ и в G_2. Обратно, пусть $\varphi \in L(G_2)$, следовательно существует некоторое дерево вывода φ в G_2. Если при этом правило $Y \to \alpha\gamma\beta$ не используется, то дерево вывода φ в G_2 является деревом вывода φ в G_1. Если же правило $Y \to \alpha\gamma\beta$ использовалось при выводе φ в G_2, то фрагмент вывода $Y \Rightarrow \alpha\gamma\beta$ заменяем на фрагмент $Y \Rightarrow \alpha X \beta \Rightarrow \alpha\gamma\beta$.

В результате получим дерево, в котором используются правила только из P, то есть получим вывод цепочки в G_1. Следовательно, из $\varphi \in L(G_2)$ следует, что $\varphi \in L(G_1)$. □

Теорема 4.2. *Пусть в грамматике G_1 имеется множество правил*
$\{Y \to \alpha X \beta, X \to \gamma_1, X \to \gamma_2, \ldots, X \to \gamma_n\}$.
Тогда, заменив это множество на множество
$\{Y \to \alpha\gamma_1\beta, Y \to \alpha\gamma_2\beta, \ldots, Y \to \alpha\gamma_n\beta, X \to \gamma_1, X \to \gamma_2, \ldots X \to \gamma_n\}$,
получим грамматику, эквивалентную G_1. И далее, если $X \neq S$ и других правил, которые имеют X в правой части нет, то группу правил $X \to \gamma_k$, $k = \overline{1,n}$ можно удалить.

Доказательство. Многократно применяя теорему 4.1 в грамматику добавляем правила $Y \to \alpha\gamma_k\beta$, где $k = \overline{1,n}$. Удаление правила $Y \to \alpha X \beta$ не приводит к потере выводимых цепочек, так как фрагмент дерева вывода $Y \Rightarrow \alpha X \beta \Rightarrow \alpha\gamma_k\beta$ можно заменить на $Y \Rightarrow \alpha\gamma_k\beta$.

Пример 4.1. Рассмотрим фрагмент грамматики для описания числа

<число> → <знак> <чбз>

<знак> → + | – | ε.

Здесь в соответствии с теоремой 4.2: *Y* - <число>, α - пустая цепочка, *X* - <знак>, β - <чбз> (число без знака), γ_1 - +, γ_2 - –, γ_3 - ε. Группу приведенных правил заменяем на правила

<число> → + <чбз> | – <чбз> | <чбз> .

Теорема 4.3. *Замена группы правил* $Y_1 \to \alpha_1 X \beta_1$, $Y_2 \to \alpha_2 X \beta_2$, ... $Y_m \to \alpha_m X \beta_m$, $X \to \gamma$ *на правила* $Y_1 \to \alpha_1 \gamma \beta_1$, $Y_2 \to \alpha_2 \gamma \beta_2$, ... $Y_m \to \alpha_m \gamma \beta_m$, $X \to \gamma$, *где других правил с нетерминалом X в левой части нет, приводит к эквивалентной грамматике. Если* $X \neq S$ *и других правил, которые имеют X в правой части нет, то правило* $X \to \gamma$ *можно удалить.*

Доказательство здесь аналогично теореме 4.2. □

Пример 4.2. Замена правил:

$S \to AB.C \mid AB. \mid A.C \mid B. \mid .C$

$A \to -$

на правила:

$S \to -B.C \mid -B. \mid -.C \mid B. \mid .C$

приводят к эквивалентной грамматике.

Теорема 4.4. Декомпозиция правил. *Замена в грамматике* G_1 *группы правил*

$$\begin{Bmatrix} Y \to \alpha_1 X \beta_1, & X \to \gamma_1 \\ \ldots\ldots & \ldots\ldots \\ Y \to \alpha_n X \beta_n, & X \to \gamma_m \end{Bmatrix} \quad \text{на группу правил} \quad \begin{Bmatrix} Y \to \alpha_1 \gamma_1 \beta_1, \ldots Y \to \alpha_1 \gamma_m \beta_1 \\ \ldots\ldots \quad \ldots \quad \ldots\ldots \\ Y \to \alpha_n \gamma_1 \beta_n, \ldots Y \to \alpha_n \gamma_m \beta_n \end{Bmatrix},$$

если $X \neq S$ *и других X в левых и правых частях правил грамматики нет, приводит к грамматике, эквивалентной* G_1.

При декомпозиции *n*+*m* правил грамматики заменяется на *n***m* правил. □

Пример 4.3. Рассмотрим КС - грамматику идентификатора, имеющую вид:

<И> → <Б> | <Б> <$И_1$>

<$И_1$> → <Б> | <Б> <$И_1$> | <Ц> | <Ц> <$И_1$>

<Б> → *a* | *b* | *c* | ... | *y* | *z*

<Ц> → *0* | *1* | *2* | ... | *8* | *9*

В предложенной грамматике *42* правила. Проведем в ней декомпозицию по <Б> и по <Ц>. Получим новую грамматику, эквивалентную заданной.

<И> → *a* | ... | *z* | *a* <$И_1$> | ... | *z* <$И_1$>

<$И_1$> → *a* | ... | *z* | *a* <$И_1$> | ... | *z* <$И_1$> | *0* | ... | *9* | *0*<$И_1$> | ... | *9*<$И_1$>

В результате получено *4*26=104* правил для букв и *2*10=20* правил для цифр, итого *124* правила. Правил стало больше, но вывод, а следовательно и разбор, будет короче. Нетрудно видеть, что рассмотренная декомпозиция позволила перейти от КС-грамматики идентификатора к А-грамматике. □

Отметим в заключении параграфа, что все рассмотренные теоремы работают в обе стороны. Так *n***m* правил при композиции можно заменить на

$n+m$ правил. Иногда лучше иметь правил поменьше и компактно описывать язык; иногда, с целью повышения эффективности разбора, их количество необходимо увеличить.

4.2. Исключение тупиковых правил из грамматик

Правило $A \to \varphi B \psi$ *называется внешним тупиком, если не существует правила* $B \to \xi$. (Из нетерминала B нет выхода).

Правило $B \to \xi$ *называется внутренним тупиком, если* $B \neq S$ *и не существует правила* $A \to \varphi B \psi$. (Нетерминал B не может появиться ни в одной сентенциальной форме, выводимой из начального символа грамматики. Внутренний тупик - это аналог недостижимого состояния конечного автомата).

Циклический тупик - это группа правил вида

$A_0 \to \varphi_1 A_1 \psi_1$

$A_1 \to \varphi_2 A_2 \psi_2$

.....

$A_n \to \varphi_0 A_0 \psi_0$,

где $A_i \neq S$ *и при этом либо для любого* A_i *не существует правил вида* $B \to \lambda A_i \mu$ *(циклический тупик внутреннего типа), либо для любого* A_i *не существует правил вида* $A_i \to \chi$ *(циклический тупик внешнего типа).* □

Теорема 4.5. *Для любой грамматики существует эквивалентная ей грамматика без тупиков.*

Для доказательства данной теоремы достаточно вспомнить, что эквивалентные грамматики порождают один и тот же язык (множество терминальных цепочек, которое выводятся из начального символа грамматики). Очевидно, что правила, включающие тупики, при порождении терминальных цепочек из начального символа грамматики просто не могут использоваться. Алгоритмы поиска тупиков и их устранения рассмотрим на примере.

Пример 4.4. Пусть задана грамматика с начальным символом S и следующим множеством правил

$S \to a\,A \mid c\,C \mid b\,D \mid q\,P \mid k\,T$ $\quad A \to c\,C$

$C \to k$ $\quad T \to m$

$D \to b\,T \mid f\,K$ $\quad P \to c\,R$

$R \to d\,K$ $\quad B \to d\,Q \mid m$

$Q \to p\,B \mid r\,B.$

Здесь выбрана автоматная грамматика только для того, чтобы наглядно проиллюстрировать ее графом состояний (рис. 4.1).

Построим новую грамматику, отбирая из исходной “хорошие” (производящие нетерминалы), из которых выводится терминальная цепочка.

Для этого положим, что множество $X_0 = V_T$,

то есть X_0 совпадает с множеством терминалов.

$X_1 = \{ A \mid \exists A \to \xi$, где $\xi \in X_o \}$,

то есть те нетерминалы, из которых терминальная цепочка получается за один шаг.

$X_2 = \{ A \mid \exists A \to \xi$, где $\xi \in (X_o \cup X_1) \}$,

.....

$X_i = \{ A \mid \exists A \to \xi$, где $\xi \in \bigcup_{j=0}^{i-1} X_j \}$,

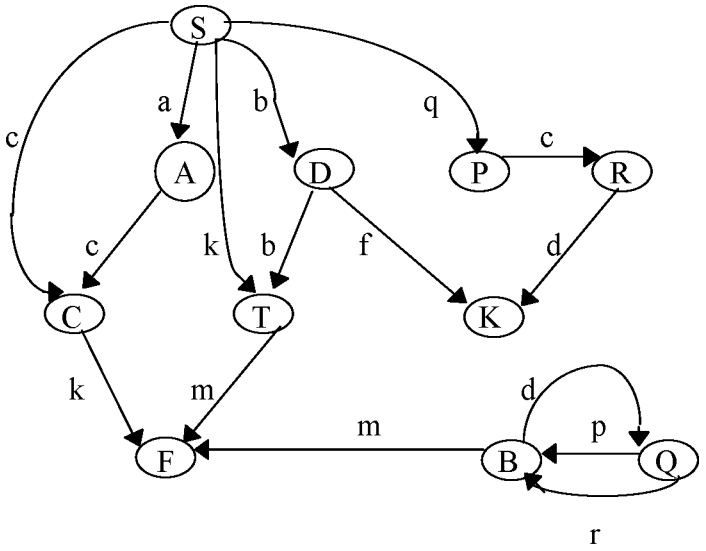


Рис. 4.1.

то есть X_i - множество нетерминалов, из которых терминальная цепочка получается не более, чем за i шагов (под шагом здесь понимается одновременная замена всех нетерминалов сентенциальной формы). На каждом шаге мы добавляем новые нетерминалы, их конечное число, следовательно конечен и данный процесс ($X_1 \subseteq X_2 \subseteq \dots X_i \subseteq V_N$).

Положим Z_i равным разности X_i и X_{i-1} ,$Z_i = X_i \setminus X_{i-1}$ - это те нетерминалы, из которых терминальная цепочка получается ровно за i шагов. Если $Z_i = \varnothing$, - пустое множество, то процесс формирования продуктивных нетерминалов пора закончить, так как из $Z_i = \varnothing$ следует, что $Z_{i+1} = \varnothing$ и т.д.

Отобранные нетерминалы обладают тем свойством, что из них выводятся терминальные цепочки. Оставшиеся нетерминалы терминальных цепочек не порождают, и их можно удалить из грамматики вместе с правилами, содержащими их в левой или правой части. Этот алгоритм позволяет устранить все внешние тупики и циклические тупики внешнего типа.

В рассматриваемом примере

$X_0 = \{a,c,b,q,k,m,f,d,p,r\}$,

$X_1 = \{C,T,B\}$, $Z_1 = \{C,T,B\}$,

$X_2 = \{S,A,C,T,D,Q,B\}$, $Z_2 = \{S,A,D,Q\}$

$X_3 = \{S,A,C,T,D,Q,B\}$, $Z_3 = \varnothing$.

Следовательно X_3 и есть то самое множество "хороших" нетерминалов. Удалив остальные нетерминалы и все ссылки на них в правилах грамматики, мы получим грамматику:

$S \to a\,A \mid c\,C \mid b\,D \mid k\,T \qquad A \to c\,C$

$C \to k \qquad\qquad T \to m$

$D \to b\,T \qquad\qquad B \to d\,Q \mid m$

$Q \to p\,B \mid r\,B.$

Для устранения внутренних тупиков повторим процесс выбора "хороших" символов с другого конца. Положим

$Y_0 = \{ S \}, \ldots, Y_i = \{ A \mid \exists B \to \varphi A \psi,$ где $A \in (V_T \cup V_N)$ и $B \in Y_{i-1} \},$

то есть Y_i - множество символов (терминалов и нетерминалов), которые могут появиться в сентенциальной форме на i - том шаге вывода.

В нашем примере

$Y_0 = \{S\},$

$Y_1 = \{a,A,c,C,b,D,k,T\},$

$Y_2 = \{c,C,k,m,b,T\}, \ Y_3 = \{k,l\}.$

Далее определим $W_i = Y_i \setminus \left(\bigcup_{k=1}^{i-1} Y_k \right) \cap V_N,$ то есть новые нетерминалы, появляющиеся на i - том шаге. Очевидно, что из $W_i = \varnothing,$ следует что процесс можно закончить, так как все нетерминалы, выводимые из начального символа грамматики, уже получены. Остальные нетерминалы и содержащие их правила грамматики можно удалить.

В нашем случае

$W_0 = \{S\},$

$W_1 = \{A,C,D,T\},$

$W_2 = \varnothing.$

Нетерминалы B и Q - внутренние тупики. Таким образом, грамматика без тупиков имеет вид:

$S \to a\,A \mid c\,C \mid b\,D \mid k\,T \qquad A \to c\,C$

$C \to k \qquad T \to m \qquad D \to b\,T\,.$

4.3. Обобщенные КС-грамматики и приведение их к удлиняющей форме

КС-грамматика называется обобщенной, если она содержит аннулирующие правила (ε - правила), то есть правила вида $A \to \varepsilon$, где ε - пустая цепочка. Обобщенная грамматика зачастую более проста и наглядна. Тем не менее следует помнить, что для любой обобщенной КС-грамматики существует эквивалентная *неукорачивающая* КС-грамматика.

Теорема 4.6. *Каждая КС-грамматика приводима к виду не более чем с одним аннулирующим правилом* $S \to \varepsilon$, *которого может и не быть.*

Доказательство. Проведем его, как обычно, конструктивно, построением неукорачивающей грамматики.

Во-первых, нужно определить, порождает ли исходная грамматика пустую цепочку. Пусть S - начальный символ исходной грамматики G. Определим в G множество нетерминалов X_i, из которых пустую цепочку можно получить за i шагов, и множество новых нетерминалов Z_i. Таким образом мы определим *аннулирующие нетерминалы*.

$X_0 = \{ A \mid \exists A \to \varepsilon \}$, $Z_0 = X_0$
$X_1 = \{ A \mid \exists A \to \xi$, где $\xi \in X_0 \}$, $Z_1 = X_1 \setminus X_0$
.....
$X_i = \{ A \mid \exists A \to \xi$, где $\xi \in \bigcup_{j=0}^{i-1} X_j \}$, $Z_i = X_i \setminus X_{i-1}$

На каком-то шаге Z_i станет равным $\varnothing$ и процесс формирования аннулирующих нетерминалов можно закончить. Если $S \notin X_j$, где $j = \overline{1,i}$, то $\varepsilon \notin L(G)$ и правила $S \to \varepsilon$ добавлять не надо. В противном случае, заменим в исходной грамматике во всех правилах S на S_1, введем новый исходный нетерминал S и к правилам грамматики G добавим правила $S \to \varepsilon \mid S_1$.

Все остальные правила вида $A \to \varepsilon$ можно удалить. Для этого заменим каждое из правил, правые части которых содержат хотя бы по одному аннулирующему нетерминалу, множеством новых правил. Если правая часть правила содержит k вхождений аннулирующих нетерминалов, то множество, заменяющее это правило, будет состоять из 2^k правил, соответствующих всем возможным способам удаления некоторых (или всех) из этих вхождений.

Пусть имеется правило
$B \to \varphi_1 A_1 \varphi_2 A_2 \varphi_3 \ldots \varphi_k A_k \varphi_{k+1}$,
где A_i $(i = \overline{1,k})$ - аннулирующие нетерминалы. Добавим к этому правилу следующие правила:
$B \to \varphi_1 \varphi_2 A_2 \varphi_3 \ldots \varphi_k A_k \varphi_{k+1}$, удалено A_1
$B \to \varphi_1 A_1 \varphi_2 \varphi_3 \ldots \varphi_k A_k \varphi_{k+1}$, удалено A_2
.....
$B \to \varphi_1 A_1 \varphi_2 A_2 \varphi_3 \ldots \varphi_k \varphi_{k+1}$, удалено A_k
$B \to \varphi_1 \varphi_2 \varphi_3 \ldots \varphi_k A_k \varphi_{k+1}$, удалены A_1, A_2
.....
$B \to \varphi_1 \varphi_2 \varphi_3 \ldots \varphi_k \varphi_{k+1}$, удалены $A_1, A_2, \ldots A_k$.

Заметим, что в случае неоднозначности, на этом шаге может получиться меньше чем 2^k правил. Так, для аннулирующего нетерминала A, правило $B \to aAA$ будет заменяться тремя правилами $B \to aAA \mid aA \mid a$, так как в данном случае безразлично первое или второе вхождение A мы рассматриваем.

После такой замены правил для всех правых частей исходной грамматики, содержащих аннулирующие нетерминалы, исключим из грамматики все ε - правила, включая те, которые могли появиться при замене.

В результате мы получим грамматику, эквивалентную исходной, что доказывается с использованием теорем 4.1 и 4.3.

Отметим, что мы рассматривали случай, когда аннулирующие нетерминалы имеют и другие альтернативы, кроме перехода в пустую цепочку. Если $A \to \varepsilon$ единственная альтернатива нетерминала A, то правые части правил, содержащие его вхождение, можно просто исключить. □

В результате применения рассмотренного алгоритма можно получить КС-грамматику, по которой вывод любой непустой цепочки характеризуется тем, что сентенциальная форма, получаемая на каждом шаге вывода, будет не короче предыдущей. Не случайно полученная грамматика носит название неукорачивающей КС-грамматики (НКС-грамматики).

Пример 4.5. Рассмотрим обобщенную КС-грамматику с аксиомой <*число*>.

<*число*> $\to$ <*знак*> <*цел.часть*> . <*др.часть*>
<*знак*> $\to$ + | - | ε
<*цел.часть*> $\to$ <*цел.часть*><*цифра*> | ε
<*др.часть*> $\to$ <*цел.часть*>
<*цифра*> $\to 0 \mid 1 \mid \ldots \mid 8 \mid 9$ и приведем ее к неукорачивающей форме.

Вначале покажем, что данная грамматика не порождает пустой цепочки. Здесь
$X_0 =$ { <*знак*>, <*цел.часть*> },
$X_1 =$ { <*др.часть*> },
$X_2 = \varnothing$ *и* $Z_2 = \varnothing$.

Среди множеств X - нет нетерминала <*число*> и, следовательно, правила <*число*> $\to \varepsilon$ добавлять не надо.

Проведем замены правил, правые части которых содержат аннулирующие нетерминалы, а затем удалим ε - правила.

В результате получим грамматику

<*число*> $\to$ <*знак*> <*цел.часть*> . <*др.часть*> | <*цел.часть*> . <*др.часть*> |
<*знак*>. <*др.часть*> | <*знак*> <*цел.часть*> . | . <*др.часть*> |
<*цел.часть*> . | <*знак*> . | .
<*цел.часть*> $\to$ <*цел.часть*><*цифра*> | <*цифра*>
<*др.часть*> $\to$ <*цел.часть*>
<*цифра*> $\to 0 \mid 1 \mid \ldots \mid 8 \mid 9$

На рис. 4.2 представлены деревья вывода цепочки +.*9* по исходной (рис. 4.2 (а)) и результирующей (рис. 4.2 (б)) грамматикам.

Для приведения грамматики к удлиняющей форме необходимо кроме аннулирующих правил исключить и цепные правила. *Цепное правило - это правило вида* $A \to B$ *, где* $A, B \in N$.

Теорема 4.7. *Для любой КС-грамматики существует эквивалентная ей грамматика без цепных правил.*

Доказательство. Пусть в грамматике имеется правило $A \to B$ и $A \neq S$ (A - не начальный символ грамматики). Тогда все правила вида $C \to \alpha A \beta$ заменим

на правила $C \to \alpha B \beta$, а правила $A \to B$ удалим. Если $A = S$ и для B существуют правила $B \to \varphi_1 | \ldots | \varphi_n$, то заменим их на $S \to \varphi_1 | \ldots | \varphi_n$, после чего $S \to B$ удалим.

Любое такое преобразование правил допустимо исходя из теорем 4.1 - 4.3 и устраняет правила вида $A \to B$. Повторяем такие преобразования до тех пор, пока в грамматике не останется цепных правил. □

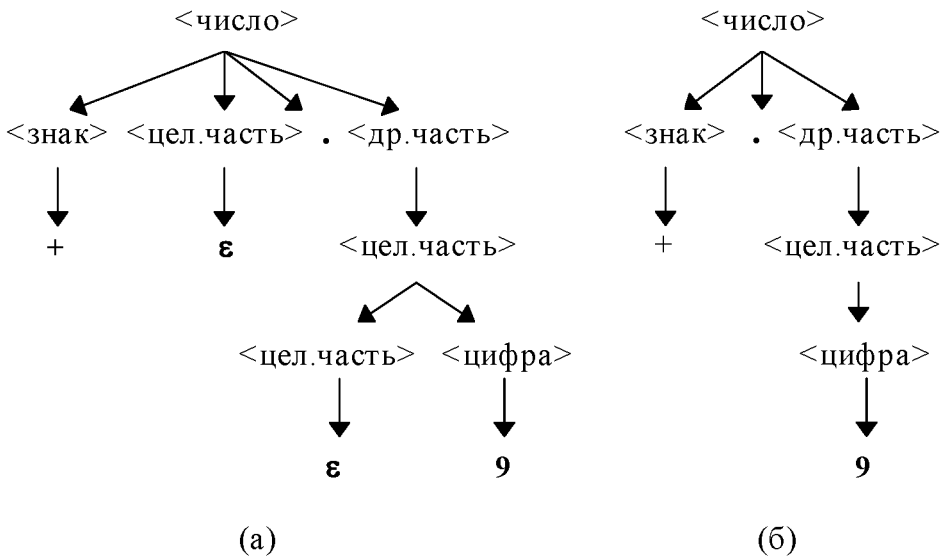


Рис. 4.2.

□

В результате устранения аннулирующих и цепных правил получается *грамматика в удлиняющей форме*, где сентенциальная форма на каждом шаге вывода будет длиннее сентенциальной формы на предыдущем шаге. Напомним, что эта форма грамматики использовалась для доказательства теоремы о разрешимости контекстных языков (теорема 1.1).

Пример 4.6. Пусть дана КС-грамматика с правилами

$S \to aBa$

$B \to A | Bc$

$A \to aA | bb$.

Правило $B \to A$ можно устранить, воспользовавшись результатами теоремы 4.2, и получить грамматику

$S \to aBa$

$B \to aA | bb | Bc$

$A \to aA | bb$ □

КС-грамматика $G=(N,\Sigma,P,S)$ называется грамматикой без циклов, если в ней нет выводов $A \Rightarrow^{+} A$ для $A \in N$. КС-грамматика G называется приведенной, если она без циклов, без аннулирующих правил и без тупиков.

Грамматики с ε- правилами и циклами иногда труднее анализировать, чем грамматики без таковых. Кроме того, в любой практической ситуации тупики (*бесполезные символы*) без необходимости увеличивают объем анализатора. Поэтому для некоторых алгоритмов синтаксического анализа, рассматриваемых

во второй части пособия, мы будем требовать, чтобы грамматики, фигурирующие в них, были приведенными. Это требование позволяет рассматривать все КС-языки.

Теорема 4.8. *Если L - КС-язык, то L=L(G) для некоторой приведенной КС-грамматики G.*

Доказательство. Применить к КС-грамматике, определяющей язык L, эквивалентные преобразования по теоремам 4.5 - 4.7. □

4.4. Устранение левой рекурсии и левая факторизация

В целом ряде приложений требуется, чтобы грамматика рассматриваемого языка не содержала левой рекурсии. Наличие леворекурсивных правил в исходной грамматике не фатально, так как *для любой КС-грамматики существует эквивалентная грамматика без левой рекурсии.*

Рассмотрим случай, когда правила грамматики *саморекурсивны, то есть левая часть правила и край правой части совпадают.* Пусть нетерминал A имеет m леворекурсивных правил $A \to A\alpha_i$ для $1 \le i \le m$ и n правил $A \to \beta_j$ для $1 \le j \le n$, которые не являются леворекурсивными (отметим, что отсутствие последних делает A тупиком).

Заменив эти правила на правила $A \to \beta_j$ <список A> для $1 \le j \le n$ <список A> $\to \alpha_i$ <список A> для $1 \le i \le m$ <список A> $\to \varepsilon$, где <список A> - новый нетерминал, мы получим эквивалентную группу правил без левой рекурсии.

Пример 4.7. Самой короткой грамматикой для представления идентификатора является леворекурсивная грамматика

<И> $\to$ *б* | <И>*б* | <И>*ц*,

где *б* - любая буква, а *ц* - любая цифра. В данной грамматике два леворекурсивных правила и одно правило без левой рекурсии.

Заменяя их на правила

<И> $\to$ *б*<$И_1$>

<$И_1$> $\to$ *б*<$И_1$> | *ц*<$И_1$> | ε,

мы получим обобщенную праворекурсивную грамматику идентификатора. Заметим, что исключение аннулирующего правила приведет нас к грамматике из примера 4.3. □

Если в грамматике имеется группа правил вида

$A \to \alpha\beta_1 | \ldots | \alpha\beta_n$,

то цепочку α можно “вынести за скобку” и преобразовать данную группу правил к виду

$A \to \alpha B$

$B \to \beta_1 | \ldots | \beta_n$.

Этот прием носит название *левой факторизации* и его необходимо знать для ряда приложений.

Упражнения к четвертой главе

4.1. В грамматике
$S \to abAcDkY$
$A \to nmDky \mid vaxYe \mid jab$
$D \to ghYo \mid kh$
$Y \to f \mid d$ устраните правила $A \to nmDky \mid vaxYe \mid jab$ и $D \to ghYo \mid kh$
и проведите декомпозицию относительно Y. Подсчитайте количество правил результирующей грамматики до проведения преобразований.

4.2. Исключите тупики из следующих грамматик:
(а)
$S \to aBcD \mid kLMp$ $L \to fM$
$B \to cLpDq \mid pDc \mid f$ $M \to Lk \mid pMLc$
$D \to fDr \mid fpq$ $K \to rF$
$F \to abc \mid ab$
(б)
$S \to AB \mid BC \mid kL$ $A \to aA \mid bL \mid c$
$B \to BS \mid AL$ $L \to cB \mid j \mid p$
$C \to QS \mid dC$ $Q \to qQ \mid aC$
$M \to xN \mid yM \mid zS \mid h$ $N \to xC \mid v \mid wM$
(в)
$S \to BA \mid CB \mid AL$ $C \to QS \mid dC$
$A \to aA \mid bB \mid cC$ $Q \to qQ \mid aC$
$B \to BS \mid AL \mid g$ $M \to xN \mid yM \mid zS \mid h$
$L \to cB \mid jC \mid p$ $N \to xC \mid v \mid wM$

4.3. Устраните аннулирующие правила из следующих грамматик:
(а)
$S \to abCDe \mid Kp$
$C \to dSde \mid \varepsilon$
$D \to dD \mid DD \mid e \mid fK \mid \varepsilon$
$K \to e \mid mcf$
(б)
$S \to aSbS \mid bSaS \mid \varepsilon$
(в)
$S \to ABC$ $A \to BB \mid \varepsilon$
$B \to CC \mid a$ $C \to AA \mid b$

4.4. Устраните цепные правила из грамматики
$E \to E{+}T \mid T$
$T \to T{*}F \mid F$
$F \to (E) \mid a$

4.5. Найдите приведенную грамматику, эквивалентную грамматике
$S \to A \mid B$ $A \to C \mid D$
$B \to D \mid E$ $C \to S \mid a \mid \varepsilon$

$D \to S \mid b \qquad E \to S \mid c \mid \varepsilon$

4.6. Устраните левую рекурсию в следующих грамматиках:

(а)

$E \to E+T \mid E-T \mid T \mid -T$

$T \to T*F \mid T/F \mid F$

$F \to (E) \mid a$

(б) $S \to AB$

$A \to Aa \mid Ab \mid d \mid e$

$B \to qK \mid rB \mid Bf \mid Bg$

$K \to vS \mid w.$

Глава 5. Свойства автоматных и контекстно-свободных языков

В этом разделе мы исследуем некоторые из основных свойств А- и КС-языков. Упомянутые здесь результаты образуют малую долю огромного богатства знаний об этих языках. Часть свойств этих языков уже были рассмотрены в разделах 1-4. Ниже мы обсудим общий вид цепочек этих языков, неоднозначность КС-грамматик и КС-языков, некоторые операции, относительно которых замкнуты классы А- и КС-языков.

5.1. Общий вид цепочек А-языков и КС-языков

Мы хотим получить характеристику цепочек А-языков, которая будет полезна для доказательства того, что некоторые языки не являются автоматными. Следующую теорему *об общем виде цепочек А-языков* называют теоремой *о "разрастании"*, потому что она в сущности говорит о том, что если дан А-язык и достаточно длинная цепочка в нем, то в этой цепочке можно найти непустую подцепочку, которую можно повторить сколько угодно раз (т.е. она "разрастается"), и все полученные таким образом "новые" цепочки будут принадлежать тому же А-языку. С помощью этой теоремы часто приводят к противоречию предположение о том, что некоторый язык является автоматным.

Теорема 5.1. *Пусть L - А-язык. Существует такая константа p, что если $\psi \in L$ и $|\psi| \geq p$, то цепочку ψ можно записать в виде $\alpha\beta\gamma$, где $0 < |\beta| \leq p$ и $\alpha\beta^i\gamma \in L$, для всех $i \geq 0$.*

Доказательство. Если L - конечный язык, то положим константу p больше длины самой длинной цепочки языка L, тогда ни одна из цепочек языка не удовлетворяет условиям теоремы и она верна. В противном случае, пусть $M = (Q, \Sigma, \delta, q_0, F)$ - конечный автомат с n состояниями и $L(M) = L$. Пусть $p = n$. Если $\psi \in L$ и $|\psi| \geq n$, рассмотрим последовательность конфигураций, которую проходит автомат M, допуская цепочку ψ. Так как в этой последовательности, по крайней мере, $n+1$ конфигурация, то найдутся две конфигурации с одинаковыми состояниями. Поэтому, должна быть такая последовательность тактов, что $(q_0, \alpha\beta\gamma) \vdash^* (q_1, \beta\gamma) \vdash^k (q_1, \gamma) \vdash^* (q_2, \varepsilon)$, для некоторого q_1 и $0 < k \leq n$. Отсюда $0 < |\beta| \leq n$.

Но тогда, для любого $i > 0$ автомат может проделать следующую последовательность тактов:

$(q_0, \alpha\beta^i\gamma) \vdash^* (q_1, \beta^i\gamma)$
$(q_1, \beta^i\gamma) \vdash^+ (q_1, \beta^{i-1}\gamma)$
.....
$(q_1, \beta^2\gamma) \vdash^+ (q_1, \beta\gamma)$
$(q_1, \beta\gamma) \vdash^+ (q_1, \gamma)$
$(q_1, \gamma) \vdash^* (q_2, \varepsilon)$.

Для случая $i = 0$ все еще очевиднее: $(q_0, \alpha\gamma) \vdash^* (q_1, \gamma) \vdash^* (q_2, \varepsilon)$

Так как $\alpha\beta\gamma \in L$, то и $\alpha\beta^i\gamma \in L$, для всех $i \geq 0$. □

Эта теорема обычно используется для доказательства того, что некоторые выбранные цепочки не являются цепочками А-языка и, следовательно, не могут быть определены А-грамматиками.

Следствие 5.1. *Язык L, состоящий из цепочек $x^n y^n$ не является автоматным языком.*

Допустим, что он автоматный. Тогда, для достаточно большого n цепочка $x^n y^n$ может быть представлена в виде $\alpha\beta\gamma$, причем $\beta \neq \varepsilon$ и $\alpha\beta^i\gamma \in L$, для всех $i \geq 0$.

Если $\beta = x...x$ или $\beta = y...y$, то $\alpha\gamma = \alpha\beta^0\gamma \notin L$, так как количество символов x и y в цепочке $\alpha\gamma$ различно. Если $\beta = x...xy...y$, то $\alpha\beta\beta\gamma = \alpha\beta^2\gamma \notin L$, так как в цепочке $\alpha\beta\beta\gamma$ символы x и y будут перемешаны. Полученное противоречие доказывает, что L - не является А-языком. □

Следствие 5.2. *Язык арифметических выражений не является А-языком*, так как он может содержать произвольное количество вложенных скобок, причем количество открывающих скобок совпадает с количеством закрывающих. Аналогично *не является А-языком любой язык, содержащий вложенные конструкции типа фигурных скобок в языке С, begin - end, repeat - until и т.п.* Каждая конечная А-грамматика, порождающая подобные конструкции, будет выводить и цепочки с неравным количеством открывающих и закрывающих скобок. Тем не менее анализировать подобные цепочки можно и с помощью автоматного подхода. При этом в синтаксисе языка допускается произвольное количество открывающих и закрывающих скобок, а контроль их парности возлагается на семантические подпрограммы. □

Прежде чем рассматривать теорему о разрастании КС-языков, примем без доказательств следующую теорему.

Теорема 5.2. *Для любой КС-грамматики, которая не допускает вывода вида $A \Rightarrow^+ \alpha A\beta$, где $|\alpha| > 0$ и $|\beta| > 0$ можно построить эквивалентную А-грамматику.* □

Иными словами любой язык, который при описании КС-грамматикой, не содержит *самовставляемых нетерминалов*, включает только одностороннюю рекурсию, при выводе наращивает цепочку в одну сторону, неважно, влево или вправо - является автоматным языком.

Теорема 5.3. *Для любого КС-языка L существует постоянная p такая, что если* $\psi \in L$ *и* $|\psi| > p$, *то* $\psi = \alpha\beta\gamma\varphi\lambda$, *где* $\beta \neq \varepsilon$, $\varphi \neq \varepsilon$ *и* $\alpha\beta^i\gamma\varphi^i\lambda \in L$ *для любого* $i \geq 0$.

Доказательство. Аналогично с теоремой 5.1 рассмотрим только случай бесконечных языков.

Рассмотрим в бесконечном КС-языке L бесповторные деревья вывода, то есть такие, у которых ни на одной ветви нет повторяющихся нетерминалов. Таких деревьев конечное число. Максимальная высота бесповторного дерева v - равна количеству нетерминалов грамматики. Если максимальная длина правых частей правил грамматики равна b, то максимальная длина цепочки, выводимой бесповторными деревьями, будет не более b^v. Положим $p = b^v$. Рассмотрим цепочку с длиной больше p и ту ветвь ее дерева вывода, в которой нетерминалы повторяются.

Рассмотрим поддеревья D_1 и D_2, начинающиеся с повторяющегося нетерминала A. Если D_1 заменить на D_2, то получим дерево вывода цепочки $\alpha\gamma\lambda$. Подвеска дерева D_2 к корню D_1 возможна, так как после нее корень дерева D_1 соответствует применению того же правила, что и корень дерева D_2. Таким образом, полученное дерево вывода является деревом вывода в той же грамматике.

Если D_2 заменить на D_1, то получим дерево вывода цепочки $\alpha\beta^2\gamma\varphi^2\lambda$. Дерево D_1, которым заменяется D_2 содержит в себе D_2 в качестве поддерева. Заменив его на D_1, получим дерево вывода цепочки $\alpha\beta^3\gamma\varphi^3\lambda$. Продолжая такие замены, можно получить любую из цепочек $\alpha\beta^i\gamma\varphi^i\lambda$. □

Пример 5.1. Пусть дана КС-грамматика с правилами:

$S \to aAp$
$A \to cAc \mid cbAb \mid d$

Максимальная высота бесповторного дерева здесь равна *2*, а максимальная длина цепочки, выводимая бесповторным деревом, равна *3* (бесповторно выводится только цепочка *adp*). На рис. 5.1 (а) показано дерево вывода цепочки *acbdbp*. Здесь принято следующее: $\alpha = a$, $\beta = cb$, $\gamma = d$, $\varphi = b$, $\lambda = p$. На рис. 5.1 (б) показана замена поддерева D_1 на D_2, а на рис. 5.1 (в) замена D_2 на D_1. □

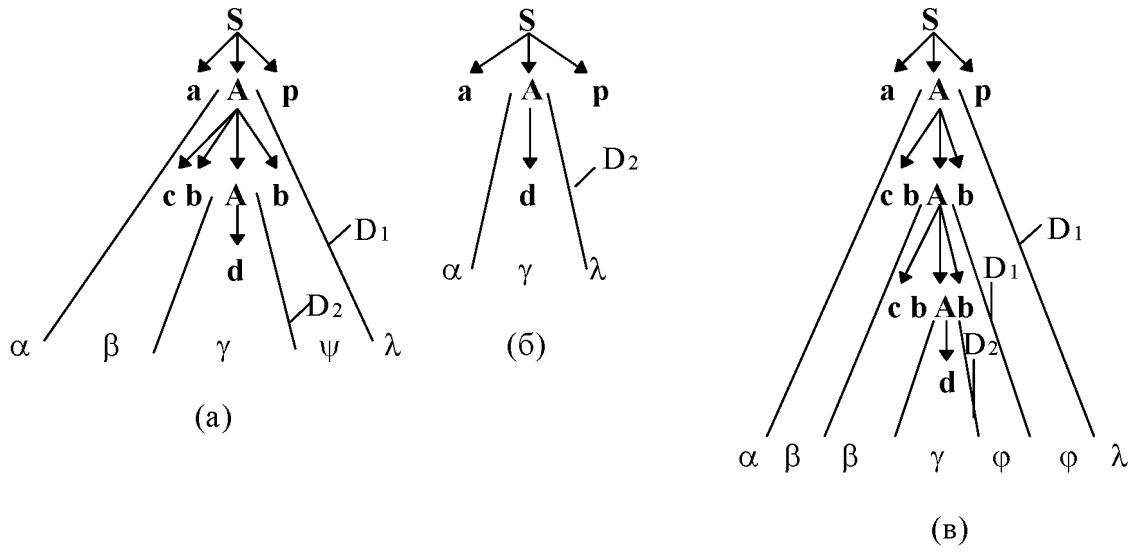


Рис. 5.1.

Теорема 5.3, как и теорема 5.1, чаще всего используется для доказательства того, что некоторые цепочки не принадлежат КС-языкам.

Следствие 5.3. *Язык L, состоящий из цепочек $x^n y^n z^n$, не является КС-языком.*

Действительно, разделяя эту цепочку на пять частей $\alpha\beta\gamma\varphi\lambda$ любым возможным способом, мы увидим, что либо $\alpha\gamma\lambda \notin L$ из-за неравного количества символов x, y и z; либо $\alpha\beta^2\gamma\varphi^2\lambda \notin L$ из-за перемешивания символов внутри цепочки.

Следствие 5.4. *Языки программирования в общем случае не являются КС-языками.*

Например, в языках программирования каждая конкретная процедура имеет одно и то же число аргументов в каждом месте, где она упоминается. Можно показать, что такой язык не контекстно-свободен, отобразив множество программ с тремя вызовами одной и той же процедуры на не КС-язык $\{0^n 10^n 10^n \mid n \geq 0\}$.

В этих языках встречаются и другие явления, характерные для не КС-языков. Так язык, требующий описания идентификаторов, длина которых может быть произвольно большой, до их использования, не контекстно-свободен. Правил КС-грамматик для описания таких явлений явно недостаточно.

Однако *на практике все языки программирования считаются КС-языками*. В компиляторах идентификаторы обычно обрабатываются лексическим анализатором и свертываются в лексемы прежде, чем достигают синтаксического анализатора. Контроль за их описанием до использования, также как и подсчет числа параметров в процедуре и т.п., возлагается на семантические подпрограммы, не входящие в собственно синтаксический анализ. Это позволяет существенно упростить синтаксис языков программирования.

5.2. Операции над языками

По определению язык - это множество цепочек, следовательно, над языками можно выполнять операции, правомерные как для множеств, так и для цепочек (строк символов). Определим некоторые из них.

Язык L называется объединением языков L_1 и L_2 ($L = L_1 \cup L_2$), если он содержит все цепочки из L_1 вместе со всеми цепочками из L_2. Формально $L = \{\alpha / \alpha \in L_1$ или $\alpha \in L_2\}$.

Язык L называется пересечением языков L_1 и L_2 ($L = L_1 \cap L_2$), если он содержит все цепочки, принадлежащие как L_1, так и L_2. Формально $L = \{\alpha / \alpha \in L_1$ и $\alpha \in L_2\}$.

Язык L называется разностью языков L_1 и L_2 ($L = L_1 \setminus L_2$), если он содержит все цепочки из L_1, которые не принадлежат L_2. Формально $L = \{\alpha / \alpha \in L_1$ и $\alpha \notin L_2\}$.

Язык L называется дополнением языка L_1 ($L = \overline{L_1}$), если он содержит все цепочки из некоторого универсального языка L_2, которые не принадлежат L_1. Формально $L = \{\alpha / \alpha \notin L_1\}$.

Язык L называется конкатенацией (сцеплением) языков L_1 и L_2 ($L = L_1 L_2$), если он содержит попарные конкатенации всех возможных цепочек из L_1 и L_2. Формально $L = \{\alpha\beta / \alpha \in L_1$ и $\beta \in L_2\}$.

Итерация языка L, обозначаемая L^*, определяется следующим образом:
(1) $L^0 = \{\varepsilon\}$,
(2) $L^n = LL^{n-1}$ для $n \geq 0$
(3) $L^* = \bigcup_{n \geq 0} L^n$.

Позитивная итерация языка L, обозначаемая через L^+, - это язык $\bigcup_{n \geq 0} L^n$.

Заметим, что $L^+ = LL^* = L^*L$ и $L^* = L^+ \cup \{\varepsilon\}$.

Язык L называется подстановкой языка L_2 в язык L_1 вместо терминала a ($L = L_{1a}^{L_2}$), если он содержит все цепочки языка L_1, в которых терминал a заменен на все возможные цепочки языка L_2.

Обращение языка L, обозначаемое L^R, - это язык, содержащий все обращенные цепочки исходного языка. Формально $L = \{\omega^R / \omega \in L\}$.

Прежде чем обсуждать практические аспекты данных определений, поговорим о том, как построить грамматику языка, полученного в результате операций над языками, и как определить ее тип.

5.2.1. Операции над КС-языками

Нетрудно показать, что целый ряд операций над КС-языками дает в результате также КС-язык.

Теорема 5.4. *КС-языки замкнуты относительно операций объединения, конкатенации, итерации, подстановки и обращения.*

Доказательство. Пусть $L_1=L(G_1)$ и $L_2=L(G_2)$ два контекстно - свободных языка, определяемых КС-грамматиками $G_1=(V_{T1},V_{N1},R_1,S_1)$ и $G_2=(V_{T2},V_{N2},R_2,S_2)$ соответственно. Проиндексируем нетерминалы грамматики G_1 индексом *1*, а нетерминалы G_2 – *2* с тем, чтобы никакие имена различных грамматик не совпадали.

Если объединить нетерминалы, терминалы, правила исходных грамматик и добавить к последнему объединению правило $S \to S_1 \mid S_2$, где S - аксиома новой результирующей грамматики, то мы, очевидно, получим КС-грамматику, определяющую объединение языков L_1 и L_2. Действительно, индексирование нетерминалов не изменяет класса исходных грамматик, точно так же как и объединение их правил и добавление контекстно-свободной продукции $S \to S_1 \mid S_2$.

Последняя продукция и обеспечивает порождение всех цепочек языка L_1 по первой альтернативе правила и всех цепочек языка L_2 по второй его альтернативе.

Таким образом $L= L_1 \cup L_2 = L(G)$, где
$G=(V_{T1} \cup V_{T2}, V_{N1} \cup V_{N2}, R=R_1 \cup R_2 \cup \{S \to S_1 \mid S_2\}, S)$ - КС-грамматика, определяющая объединение языков L_1 и L_2.

Точно также можно показать, что КС-грамматика
$G=(V_{T1} \cup V_{T2}, V_{N1} \cup V_{N2}, R=R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S)$ определяет конкатенацию языков L_1 и L_2 ($L(G)= L_1 L_2$).

Если к правилам P_1 исходной грамматики G_1 добавить правило $S \to SS_1 \mid \varepsilon$, считая S начальным символом новой КС-грамматики G, то грамматика G будет определять итерацию языка L_1 - L_1^* . Если же к P_1 добавить правило $S \to SS_1 \mid S_1$, или правило $S \to S_1 S \mid S_1$, или правило $S \to SS \mid S_1$, то полученная КС-грамматика G будет определять позитивную итерацию L_1^+.

Если во всех правилах грамматики G_1 вида $A \to \varphi a \psi$ заменить терминал a на S_2 - аксиому грамматики G_2, то полученная в результате таких преобразований КС-грамматика G будет определять не что иное, как язык L - подстановку языка L_2 в язык L_1 вместо терминала a:
$L = L_{1a}^{L_2}\ G=<V_{T1} \cup V_{T2} \backslash \{a\}, V_{N1} \cup V_{N2}, R=R_1 \cup R_2 \cup \{S \to \alpha S_2 \beta\} \backslash \{S_1 \to \alpha\, a \beta\}, S>$

Для того, чтобы получить грамматику, определяющую обращение L^R для исходного языка $L(G)$ достаточно обратить левые и правые части правил исходной грамматики G, то есть правила $\alpha \to \beta$ заменить на правила $\alpha^R \to \beta^R$ (в КС-грамматиках правила $A \to \beta$ заменяются на правила $A \to \beta^R$). Такие преобразования не изменяют класса КС-грамматик и позволяют порождать все обращенные цепочки. □

Все рассмотренные преобразования КС-грамматик достаточно очевидны. Рассмотрим на примере только формирование грамматики для обращения языка.

Пример 5.2. Пусть задана грамматика с правилами
$S \to aS \mid bB$
$B \to cB \mid d$

Для простоты здесь взята А-грамматика, но ничто не мешает рассматривать ее как КС-грамматику. Цепочки, порождаемые данной грамматикой состоят из необязательных символов “*a*” в начале цепочки, символа “*b*”, затем необязательных символов “*c*” и символа “*d*” в конце, т.е. цепочка имеет вид

[aaa...]b[ccc...]d,

где квадратные скобки традиционно обозначают необязательный элемент.

Обратим правила заданной грамматики и в результате получим:

$S \to Sa \mid Bb$

$B \to Bc \mid d$

Если правила исходной грамматики обеспечивали вывод цепочки слева направо, то полученные правила выводят ее справа налево. Цепочка, порождаемая последней грамматикой имеет вид *d[ccc...]b[aaa...]*. □

Теорема 5.5. *КС-языки не замкнуты относительно операций пересечения, дополнения и разности.*

Доказательство. Языки $L_1=\{a^n b^n c^j \mid n \geq 1, j \geq 1\}$ и $L_2=\{a^j b^n c^n \mid n \geq 1, j \geq 1\}$ - КС-языки. Первый из них можно определить правилами

$S \to XY$

$X \to aXb \mid ab$

$Y \to cY \mid c$,

а второй

$S \to XY$

$X \to aX \mid a$

$Y \to bYc \mid bc$.

Однако $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$ - не КС-язык (см. следствие теоремы 5.3). Таким образом, класс КС-языков не замкнут относительно пересечения.

Отсюда можно также заключить, что класс КС-языков не замкнут относительно дополнения. В силу закона де Моргана $(\overline{A \cup B} = \overline{A} \cap \overline{B})$ любой класс языков, замкнутый относительно объединения и дополнения, должен быть замкнут относительно пересечения. Из теоремы 5.4 известно, что КС-языки замкнуты относительно объединения и предположение об их замкнутости относительно дополнения приводит нас к противоречию с первым доказанным положением данной теоремы.

Для доказательства последнего положения достаточно вспомнить, что дополнение - это, по сути дела, разность множеств. □

5.2.2 Операции над А-языками

Теорема 5.6. *Автоматные языки замкнуты относительно операций объединения, конкатенации, итерации, обращения, подстановки, пересечения, дополнения и разности.*

Доказательство. Проведем его конструктивно, также как и в теореме 5.4. Для представления А-грамматик используем графы состояний и в случае операций над двумя языками индексируем нетерминалы исходных грамматик.

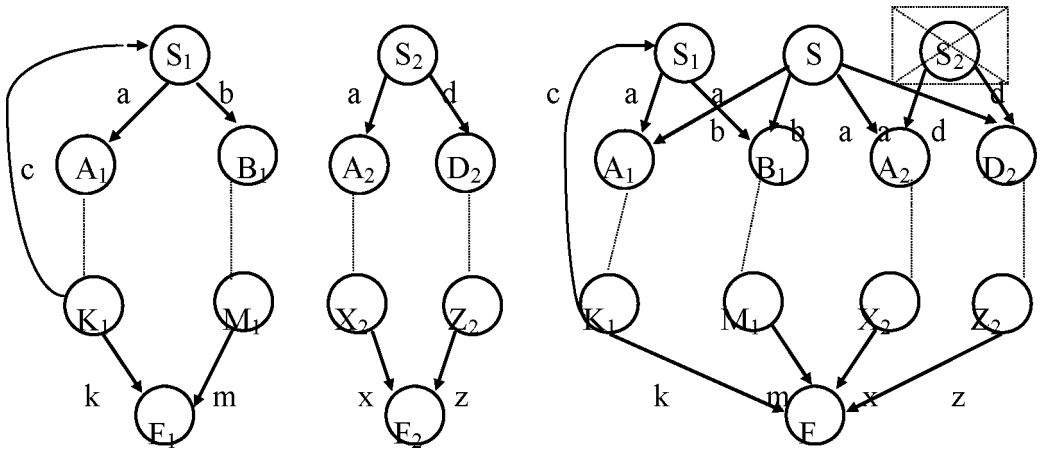
Объединение. Пусть даны два А-языка $L_1=L(G_1)$ и $L_2=L(G_2)$ и графы состояний грамматик G_1 и G_2, схематично представленные на рисунках 5.2 (а) и (б), соответственно.

На рисунке 5.2 (в) представлена грамматика G, определяющая объединение исходных языков. Для ее построения вводим новый начальный символ S. Если в исходных грамматиках из S_i в A_i ведет ребро, помеченное терминалом a, то проведем ребро из S в A_i и пометим его тем же терминалом a. Выберем новый конечный символ F и все ребра, шедшие в F_1 и F_2 проведем в F, а F_1 и F_2 удалим. Вершины S_1 и S_2 в общем случае удалять нельзя, так как к ним могут идти ребра, но если в S_i возвратов нет, то эту вершину (нетерминал) можно удалить (в нашем примере можно удалить вершину S_2 вместе с выходящими из нее дугами).

Очевидно, что результирующая грамматика G является А-грамматикой. Зачастую она может быть недетерминированной, но перевод А-грамматики из недетерминированной формы в детерминированную уже был рассмотрен ранее.

Конкатенация. В этом случае получение грамматики-результата сводится к склеиванию начальной вершины S_2 языка-суффикса с заключительной вершиной F_1 языка-префикса, т.е. все ребра, шедшие в F_1 направляются в S_2, а F_1 удаляется (см. рис. 5.3 (а)).

Итерация. Для каждого ребра, идущего из некоторой вершины A исходной грамматики в заключительную вершину F, строится дублирующее его ребро, ведущее из A в начальную вершину S. На рис. 5.3 (б) добавляемые ребра выделены жирной линией.



(а) (б) (в)

Рис. 5.2.

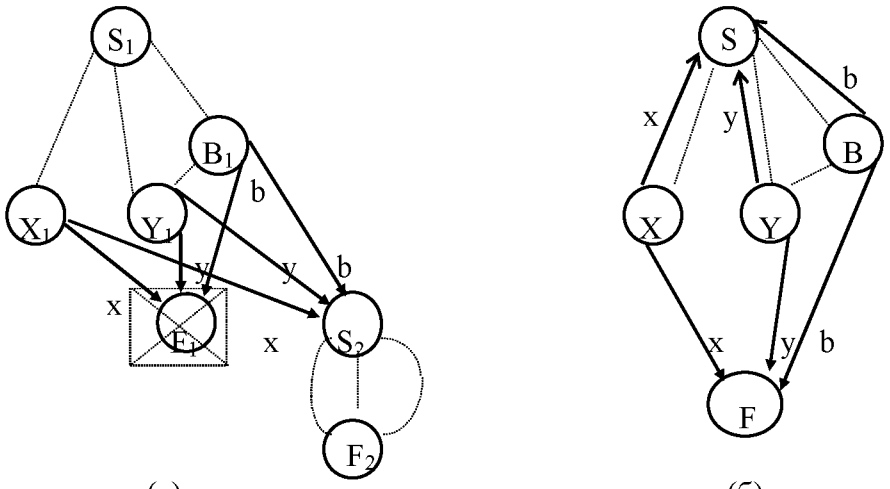


Рис. 5.3.

Обращение. На рис. 5.4 (а) представлен граф исходной грамматики. Изменим имя начальной вершины S на S_1 и добавим вершину S_2. Для всех ребер выходящих из S_1 и входящих в A добавим дуги, выходящие из S_2 и входящие в A (см. рис. 5.4 (б)). Заменим имя заключительной вершины F на имя начальной - S, а имя вершины S_2 на имя заключительной - F и изменим ориентацию ребер. В результате мы получим А-грамматику, определяющую обращение исходного языка. Граф этой грамматики представлен на рис. 5.4 (в).

Заметим, что добавление вершины необходимо только в случае возвратов в начальную вершину исходной грамматики. Если возвратов нет, то достаточно изменить ориентацию ребер и сделать перестановку имен начального и заключительного состояний.

Подстановка. На рис 5.5 (а) представлена грамматика G_2 языка L_2, который мы хотим подставить вместо терминала a в язык L_1 с грамматикой G_1, приведенной на рис. 5.5 (б). Возьмем столько экземпляров G_2, сколько в G_1 имеется ребер, помеченных терминалом a. Нетерминалы в G_1 отметим индексом 0, а нетерминалы в i - ом экземпляре G_2 индексом i. На место каждого ребра G_1, помеченного терминалом a и идущего из A_0 в B_0, подставим экземпляр G_2, т.е. вершину A_0 из G_2 совместим с вершиной S_i , а вершину B_0 - с вершиной F_i. Отметим, что при наличии возвратов в начальную вершину грамматики G_2 и других ребер, идущих из A_0 грамматики G_1 и помеченных терминалами, отличными от a, необходимо расщеплять начальную вершину грамматики G_2 на две вершины. Одна из них в точности совпадает с исходной, а другая повторяет все выходы исходной начальной вершины, но возвраты в нее опускаются.

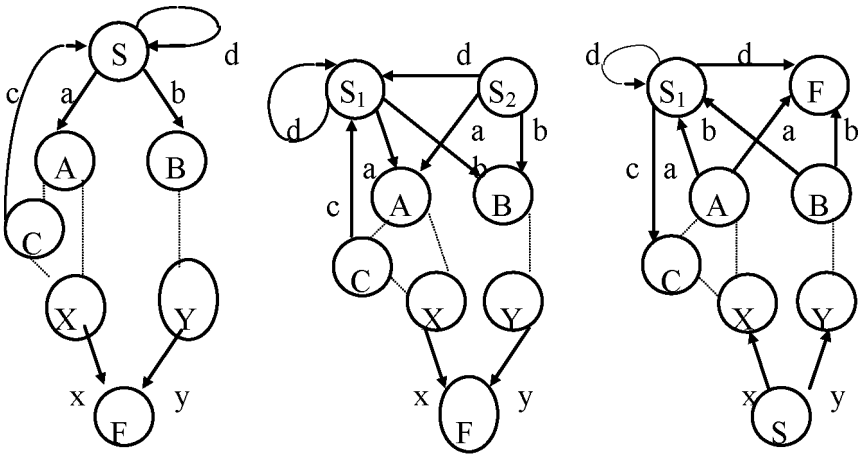


Рис. 5.4.

Именно эту, вторую начальную вершину без возвратов и совмещают с A_0. Результаты этих преобразований приведены на рис. 5.5. (в), отражающем грамматику языка, полученного в результате указанной подстановки.

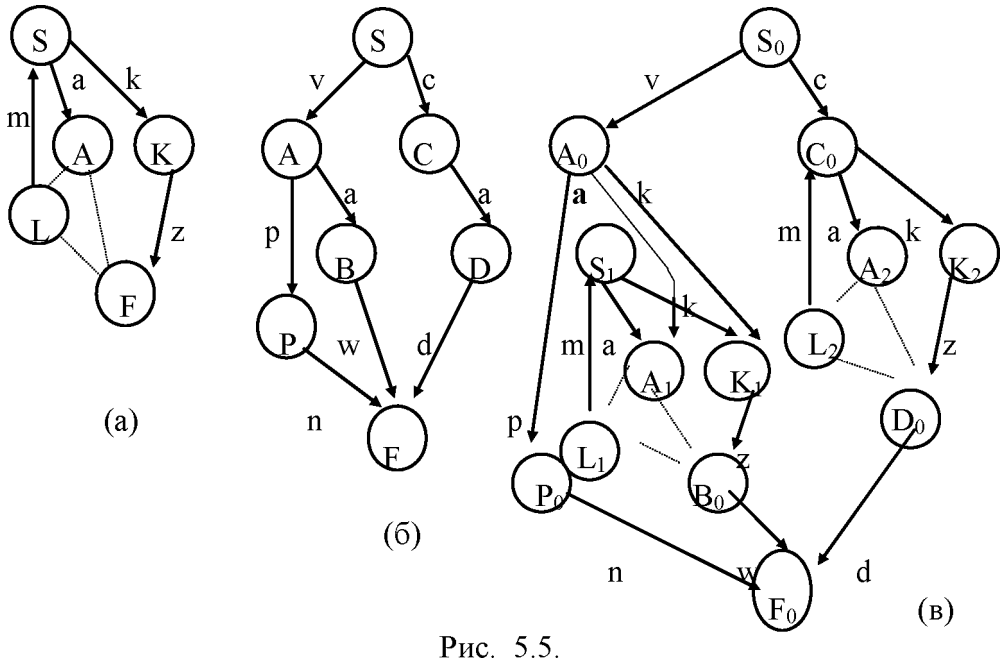


Рис. 5.5.

Пересечение. Здесь мы отойдем от принятого выше представления А-грамматик в виде графов состояний и рассмотрим построение грамматики, определяющей пересечение двух А-языков на конкретном примере.

Пример 5.3. Пусть А-язык L_1 определяется А-грамматикой $G_1 = (V_{T1}, V_{N1}, R_1, S_1)$ и множество R_1 - это группа модифицированных правил

$S \to aS \mid bC \mid dC$

$C \to bC \mid cC \mid \perp\!\!\!\perp F$,

где F - заключительный нетерминал, и А-язык L_2 определяется А-грамматикой $G_2 = (V_{T2}, V_{N2}, R_2, S_2)$ и

$$R2=\left\{\begin{matrix} R\to aQ|bM \\ M\to mM|nQ \\ Q\to bQ|kQ|\perp F \end{matrix}\right\}$$

Выполним формальную процедуру операции пересечения.

Определим грамматику $G=(V_N, V_T, R, <SR>)$ языка $L = L_1 \cap L_2$. Для того, чтобы проконтролировать наше решение вначале определим вид цепочек, как заданных языков, так и языка - результата, благо простота выбранных грамматик позволяет легко это сделать. Цепочки языка L_1 могут содержать в начале произвольное количество символов a, обязательный символ b или d, затем, возможно, серию символов b и (или) c и в завершении символ $\perp$. Схематично цепочку языка L_1 можно представить в виде $[a...a]b\begin{bmatrix} b...b \\ c...c \end{bmatrix}\perp$, где квадратные скобки ограничивают необязательные части строки, многоточие обозначает произвольное количество символов, а две строки - произвол в выборе символов. Цепочки языка L_2 имеют вид $a\begin{bmatrix} b...b \\ k...k \end{bmatrix}\perp$ или $b[m...m]n\begin{bmatrix} b...b \\ k...k \end{bmatrix}\perp$, а цепочка результирующего языка - $ab[b...b]\perp$.

Заметим, что $\Sigma \subseteq \Sigma_1 \cap \Sigma_2$. Построение грамматики-пересечения напоминает построение детерминированной формы А-грамматики. В качестве элементов нового множества нетерминалов выбираются пары нетерминалов исходных грамматик типа $<SR>$, $<SQ>$, $<SM>$, $<CM>$, $<CQ>$ и т.п. В результате построения правил грамматики-пересечения часть этих нетерминалов может быть исключена, как внутренние или внешние тупики. Схема построения правил новой грамматики состоит в том, что рассматриваются только те пары нетерминалов и те их альтернативы, которые имеют одни и те же терминалы в качестве продолжения цепочки. В результате мы получим грамматику

$<SR> \to a<SQ> \mid b<CM>$

$<SQ> \to b<CQ>$

$<CQ> \to b<CQ> \mid \perp <FF>$.

Заметим, что нетерминал $<CM>$ не имеет общего продолжения, является внешним тупиком и его можно исключить вместе с правилом $<SR> \to b<CM>$.

То есть операция пересечения $L=L_1 \cap L_2$ определяется следующим образом:

$G=<V_{T1} \cap V_{T2}, V_N=\{<A_1A_2>, A_1 \in V_{N1}, A_2 \in V_{N2}\}, <S_1S_2>, R=\{<AB> \to a<CD>$, если в исходной грамматике G_1 присутствует правило вида $A \to aC$: $A,C \in V_{N1}$, $B \to aD$, $B,D \in V_{N2}\}>$.

В результате такого построения получается язык, включающий множество цепочек, принадлежащих языку L_1 и L_2. Действительно:

а) если $\varphi \in L_1, L_2 \Rightarrow$ существует вывод в L_1 и $L_2 \Rightarrow$ в L_1: $\varphi = ab...f$, значит существует вывод $S_1aA_1bB_1...fF_1$ в G_1 и вывод в G_2: $S_2aA_2bB_2...fF_2$.

По построению, если существуют такие правила вывода, то в L появится правило вида $<S_1S_2>a<A_1A_2>...f<F_1F_2>$.

Значит, если есть такой вывод, то цепочка φ принадлежит L.

б) Проводя аналогичные рассуждения в обратном порядке, получим, что любая цепочка, принадлежащая языку L, принадлежит языкам L_1 и L_2. □

Рассмотрим еще один пример:

$$G_1: \begin{cases} S \to aB \\ B \to cF \mid dF \end{cases}$$

$$G_2: \begin{cases} S \to aK \mid cK \\ K \to cF \end{cases}$$

В результате выполнения операции пересечения получим:

$$\begin{cases} < SS > \to a < BK > \\ < BS > \to c < FK > = \text{тупик} \\ < BK > \to c < FF > \end{cases}$$

5.2.3. Операции над контекстными языками

Теорема. *К-языки, также как и КС-языки, замкнуты относиельно операций объединения, конкатенации, итерации, обращения, подстановки и не замкнуты относительно операций пересечения, дополнения и разности.*

Доказательство:

Алгоритмы выполнения перечисленных операций аналогичны алгоритмам для КС-языков, кроме операции конкатенации и подстановки.

До выполнения операций конкатенации и подстановки в К-языках выполняется преобразование полутерминалов. Суть этого преобразования состоит в следующем: каждый из терминальных символов исходных грамматик заменяется на нетерминал, помеченный соответствующим терминальным символом: например, a заменяется на Aa и добавляются правила вида $Aa \to a$. После этого осуществляется индексация нетерминалов исходных грамматик, что приводит к тому, что терминальные символы в разных грамматиках помечаются разными индексами и могут быть получены только в самом конце вывода цепочки, в результате этого контекст исходных грамматик не может привести к выводу цепочек, не принадлежащих результирующему языку.

Например: $G1$: $S \to aA$ $\quad$ $G2$: $S \to b$

$Ab \to cb$

$A \to a$

Выполним операцию конкатенации без преобразования полутерминалов, проиндексировав нетерминалы исходных грамматик:

$S \to S_1S_2$

$S_2 \to ab$

$S_1 \to aA_1$

$A_1b \to cb$

$A_1 \to a.$

В результате может быть выведена цепочка – acb, не принадлежащая языку. Теперь выполним преобразование полутерминалов и только затем проиндексируем грамматики, в результате чего цепочки, не принадлежащие языку, в грамматике не выводимы:

$S \to S_1 S_2$

$S_2 \to B b_2$ $\quad S_1 \to A a_1 A_1$

$A_1 \to a_1$; $A_1 B b_1 \to C c_1 B b_1$

$B b_2 \to b$ $\quad A a_1 \to a$; $B b_1 \to b$; $C c_1 \to c$

5.3. Выводы для практики

В результате операций над языками получается новое множество цепочек, то есть новый язык, который “собран из кусков”, состоит из частей исходных языков. С практической точки зрения очень важно, что язык можно разложить на более простые языки и формировать сложный язык на базе простых языков и операций над ними. Так, если нам известен язык букв $L_б$ (любая латинская буква) и язык цифр $L_ц$, то можно совершенно тривиально определить язык идентификаторов $L_и$, конкатенацию языка букв и итерацию объединения языка букв и языка цифр: $L_и = L_б\,(L_б \cup L_ц)^*$.

Отметим, что при реализации на ЭВМ программы синтаксического анализа различных языков лучше всего представлять в виде логических процедур-функций, дающих положительный ответ в случае принадлежности цепочки или ее фрагмента рассматриваемому языку и отрицательный при обнаружении ошибок. При таком построении итерация языка подменяется вызовом соответствующей процедуры внутри цикла; конкатенация - последовательным вызовом процедур анализа составляющих языков; объединение - альтернативным вызовом процедур; подстановка - заменой анализа терминала на процедуру анализа подставляемого языка.

Пример 5.4. Рассмотрим простейший язык записи результатов шахматного матча для одного из его участников, цепочки которого имеют вид:
+ победы = ничьи - поражения;
где *победы, ничьи* и *поражения* представляются в виде целых констант без знака или имен переменных (идентификаторов). Данный язык

$L = L_+ L_{блок} L_= L_{блок} L_- L_{блок} L_;$,

где $L_+ = \{+\}$, $L_= = \{=\}$, $L_- = \{-\}$, $L_{блок} = L_и \cup L_к$, $L_и = L_б\,(L_б \cup L_ц)^*$, $L_к = L_ц^+$, $L_б = \{a,b,...,y,z\}$, $L_ц = \{0,1,...,9\}$, $L_; = \{;\}$.

5.4. Неоднозначность КС-грамматик и языков

Напомним, что *КС-грамматика G неоднозначна, если существует цепочка $\alpha \in L(G)$, имеющая два или более различных деревьев 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. Неоднозначность - нежелательное свойство КС-грамматик и языков.

Пример неоднозначной КС-грамматики арифметических выражений был рассмотрен в разделе 1.1. Но самый известный пример неоднозначности в языках программирования - это “кочующее *else*”.

Пример 5.5. Рассмотрим грамматику с правилами вывода

$S \to if\ b\ then\ S\ else\ S \mid if\ b\ then\ S \mid a$

Эта грамматика неоднозначна, так как цепочка

if b then if b then a else a

имеет два дерева вывода, первое из которых (рис 5.6. (а)) предполагает интерпретацию

if b then (if b then a) else a ,

а второе (рис 5.6 (б))-

if b then (if b then a else a)

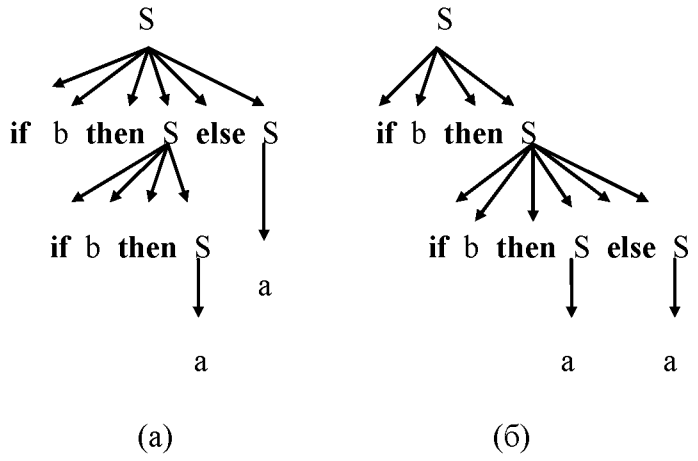


Рис 5.6.

Хотелось бы иметь алгоритм, который по произвольной КС-грамматике выяснял, однозначна она или нет. Но, к сожалению, можно доказать, что *проблема, однозначна ли КС-грамматика G, алгоритмически неразрешима.* Хотя такого алгоритма нет, можно указать некоторые встречающиеся в правилах конструкции, приводящие к неоднозначности, которые можно распознать на практике и избегать при описании языков программирования.

Грамматика, содержащая правила $A \to AA \mid \alpha$, неоднозначна, так как подцепочка *AAA* допускает два различных разбора (рис. 5.7).

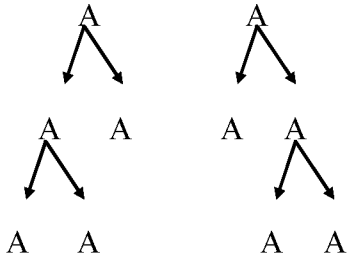


Рис. 5.7.

Здесь можно устранить неоднозначность, если вместо предложенных правил с двухсторонней рекурсией использовать одностороннюю, то есть использовать правила $A \to AB \mid B$ и $B \to \alpha$ или правила $A \to BA \mid B$ и $B \to \alpha$.

Другой пример неоднозначности - правило $A \to A\alpha A$, так как цепочку $A\alpha A\alpha A$ можно получить по двум разным деревьям вывода. Пара правил $A \to \alpha A \mid A\beta$ тоже создает неоднозначность, - цепочка $\alpha A\beta$ имеет два разных левых вывода $A \Rightarrow \alpha A \Rightarrow \alpha A\beta$ и $A \Rightarrow A\beta \Rightarrow \alpha A\beta$.

Все перечисленные примеры, так или иначе, связаны с двухсторонней рекурсией. Более тонкий пример - пара правил $A \to \alpha A \mid \alpha A\beta A$, по которым цепочка $\alpha\alpha A\beta A$ имеет два вывода $A \Rightarrow \alpha A\beta A \Rightarrow \alpha\alpha A\beta A$ и $A \Rightarrow \alpha A \Rightarrow \alpha\alpha A\beta A$. Если при двухсторонней рекурсии средством борьбы с неоднозначностью является устранение рекурсии с одной из сторон, то в последнем случае поможет левая факторизация.

Из приведенных примеров ясно, что определенная выше неоднозначность - это свойство грамматики, а не языка. Для некоторых неоднозначных грамматик можно построить эквивалентные им однозначные грамматики.

Пример 5.6. Рассмотрим грамматику из примера 5.5. Эта грамматика неоднозначна потому, что *else* можно ассоциировать с двумя различными *then*. Неоднозначность можно устранить, если связать *else* с последним из предшествующих ему *then*, как на рис. 5.6 (б). Для этого введем два нетерминала S_1 и S_2 с тем, чтобы S_2 порождал только полные операторы вида *if-then-else*, а S_1 - операторы обоих видов. Правила новой грамматики имеют вид

$S_1 \to$ *if b then* $S_1 \mid$ *if b then* S_2 *else* $S_1 \mid a$

$S_2 \to$ *if b then* S_2 *else* $S_2 \mid a$

Тот факт, что слову *else* предшествует только S_2, гарантирует появление внутри конструкции *then-else* либо символа *a*, либо другого *else*. Таким образом, структура, изображенная на рис. 5.6 (а), здесь не возникает. □

КС-язык называется неоднозначным (или существенно неоднозначным), если он не порождается никакой однозначной КС-грамматикой.

С первого взгляда не видно, существуют ли вообще неоднозначные КС-языки, но нашим следующим примером и будет такой язык.

Пример 5.7. Пусть $L = \{a^i b^j c^k \mid i = j$ или $j = k\}$. Этот язык неоднозначен, что можно строго доказать. Интуитивно же это объясняется тем, что цепочки с $i = j$ должны порождаться группой правил, отличных от правил, порождающих цепочки с $j = k$. Тогда, по крайней мере некоторые из цепочек с $i = j = k$ должны порождаться обеими механизмами. Одна из КС-грамматик, порождающих L, такова:

$S \to AB \mid DC$

$A \to aA \mid \varepsilon$

$B \to bBc \mid \varepsilon$

$C \to cC \mid \varepsilon$

$D \to aDb \mid \varepsilon$

Ясно, что она неоднозначна и на рис. 5.8 представлены два дерева вывода цепочки *aabbcc*. Проблема, порождает ли данная КС-грамматика однозначный

язык (т.е. существует ли эквивалентная ей однозначная грамматика), алгоритмически неразрешима. Но для некоторых больших подклассов КС-языков известно, что они однозначны. Именно к этим подклассам и относятся все созданные до сих пор языки программирования.

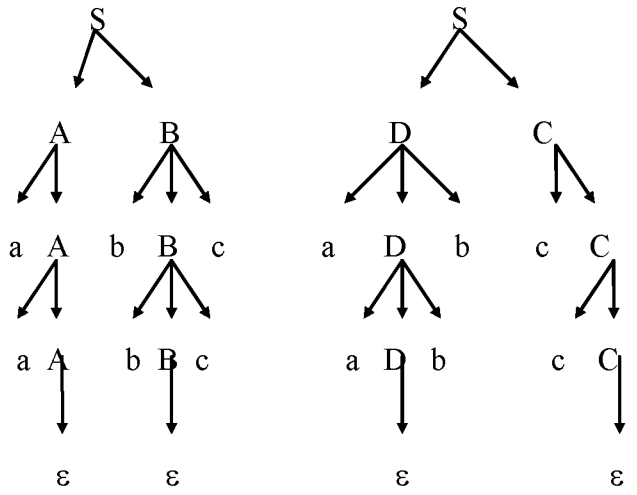


Рис. 5.8.

Упражнения к пятой главе

5.1. Покажите, что следующие языки не являются А-языками:
(а) $\{0^n\ 1\ 0^n \mid n \geq 1\}$,
(б) $\{\omega\omega \mid \omega \in \{0,1\}^*\}$
(в) $L(G)$, где G определено правилами $S \to aSbS \mid c$.

5.2. Покажите, что следующие языки не контекстно-свободны:
(а) $\{a^i\ b^i\ c^j \mid j \leq i\}$;
(б) $\{a^i\ b^j\ c^k \mid i < j < k\}$;
(в) $\{a^i\ b^j\ a^j\ b^i \mid j \leq i\}$;
(г) $\{a^m\ b^n\ a^m\ b^n \mid m, n \geq 1\}$;
(д) $\{a^i\ b^j\ c^k \mid$ все числа i, j, k разные$\}$

5.3. Пусть имеется следующая серия непересекающихся языков: $L_{ми}$ - язык мужских имен - *{Миша, Андрей, Петя, ... }*, $L_{жи}$ - язык женских имен - *{Марина, Таня, Катя, ... }*, $L_{си}$ - язык "средних" имен - *{Валя, Саша, Женя, ... }*, $L_{мф}$ - язык мужских фамилий - *{Шамашов, Сидоров, Петров, ... }*, $L_{жсф}$ - язык женских фамилий - *{Наянова, Михеева, Соловьева, ... }* и язык "средних" фамилий $L_{сф}$ = *{Кричевер, Пономаренко, Перебейнос, ... }*. Требуется с помощью операций над заданными языками построить язык L - всех правильных сочетаний имен и фамилий. Попробуйте оптимизировать вашу запись.

5.4. С помощью операций над языками постройте язык L - правильных процедур Паскаля, если имеются: язык заголовков, начала и концов процедур L_p, L_{begin}, L_{end}; языки описания переменных L_{var}, L_{const}, L_{type}; языки операторов присваивания $L_{:=}$; ветвлений - L_{if}, L_{case}; циклов L_{while}, L_{for} и т.п. При этом

считается, что перечисленные языки охватывают всю рассматриваемую конструкцию.

5.5. Пусть имеется язык заголовков цикла - L_{while}, языки начала и конца блоков L_{begin} и L_{end}, язык оператора присваивания $L_{:=}$. Постройте язык циклов L, считая, что тело цикла могут составлять операторы присваивания, вложенные или/и чередующиеся циклы данного типа.

5.6. Постройте грамматики объединения, итерации, конкатенации, обращения, подстановки и пересечения языков со следующими правилами грамматики:

G_1: $S \to aA \mid bB \mid c$ $\quad$ G_2: $S \to aA$

$A \to aS \mid a$ $\quad$ $A \to bA \mid bC$

$B \to b$ $\quad$ $C \to kA \mid cS \mid d$

(а) рассматривая исходные языки как КС-языки,

(б) рассматривая их как А-языки.

Глава 6. Синтаксический анализ КС-языков

6.1.Методы анализа КС-языков. Грамматики предшествования

В А-грамматике синтаксический анализ цепочки прост: начиная с первого символа цепочки, переходя из состояния в состояние, делается вывод о возможности получения очередного символа цепочки либо о переходе автомата в ошибочное состояние. Для вывода цепочки *aaaa* в грамматике $S \to aS \mid a$ необходимо выполнить следующую последовательность действий: $aS \to aaS \to aaaS \to aaaa$, предварительно приведя грамматику к детерминированной форме. В КС-грамматиках используется дерево вывода и грамматики также могут быть неоднозначными.

Большинство языков программирования можно описать с помощью контекстно-свободных грамматик. Но построить алгоритм анализа по таким грамматикам в общем случае не так просто и эти алгоритмы, как правило, неэффективны. Один из путей анализа цепочек КС-языка следует из теоремы о разрешимости таких языков. Напомним, что любой КС-язык можно описать с помощью КС-грамматики в удлиняющей форме, по которой любая цепочка заданного языка длины n выводится не более чем за n шагов. Для того чтобы определить принадлежность некоторой цепочки с длиной n данному языку, необходимо по заданной грамматике вывести все бесповторные цепочки, которые выводятся не более чем за n шагов, и сравнить их с исходной цепочкой. Если мы обнаружим совпадение, то исходная цепочка принадлежит языку, в противном случае – не принадлежит. Совершенно очевидно, что подобный потенциально осуществимый алгоритм не имеет практического смысла. Во-первых, необходимо строить вывод миллиардов цепочек, во-вторых, такой алгоритм может ответить только на вопрос принадлежности языку. Локализовать и идентифицировать ошибку, а тем более осуществить трансляцию в процессе синтаксического анализа текста он не способен.

Методы построения анализаторов по КС-грамматике делятся на две группы: нисходящие (анализ сверху вниз) и восходящие (анализ снизу вверх). Эти термины соответствуют способу построения синтаксических деревьев вывода.

Нисходящие методы выполняют анализ с начального выделенного нетерминала, или иначе от корня дерева вывода вниз, к концевым вершинам, пытаясь построить дерево вывода цепочки. Если дерево удалось построить, то цепочка принадлежит языку, в противном случае – не принадлежит. В общем случае все методы основаны на переборе всех возможных вариантов вывода, поэтому неэффективны, хотя и существуют алгоритмы нисходящего разбора с возвратами, ограничивающих число переборов вариантов при выводе цепочек.

Алгоритмы восходящего разбора начинаются с листьев дерева вывода (с входных символов, как и в автоматных грамматиках) и пытаются построить дерево разбора, восходя от листьев к корню. Среди этих алгоритмов наиболее эффективными являются алгоритмы, использующие при свертке отношения предшествования.

В цепочке КС-языка сложно выделить основу для свёртки. Впервые в 1966 г. Вебером был предложен метод свёртки, используя отношение предшествования. Согласно этому методу, между любыми символами, которые могут встретится на любом шаге вывода цепочки, определяется одно из трёх отношений предшествования:

$\doteq$ – отношение предшествования равенства,

отношение предшествования меньше (раньше): $<\cdot$

и отношение предшествования больше (позже): $\cdot >$

Между любыми двумя символами S_1 и S_2 устанавливается отношение $\doteq$, если они находятся рядом в одном правиле вывода.

$S_1 <\cdot S_2$, если S_1 уже выведено, а S_2 – будет стоять рядом с S_1 при применении некоторого правила на следующем шаге вывода.

$S_1 \cdot > S_2$ (S_1 “позже” S_2), если S_2 выведено, а S_1 будет стоять рядом с S_2 на следующем шаге вывода.

Основа для свёртки – множество символов, находящихся между отношениями предшествования $<\cdot$ и $\cdot>$.

Наиболее эффективные методы свертки применимы для узкого класса КС-грамматик, называемых грамматиками предшествования, для которых выполняются следующие ограничения.

Грамматики предшествования – узкий класс КС-грамматик, в которых:
1) между любыми символами существует не более одного отношения предшествования;
2) не существует правил с одинаковыми правыми частями.

В классе грамматик предшествования существует несколько видов грамматик, различающихся правилами определения отношений предшествования между символами. Существуют грамматики простого и расширенного предшествования, в зависимости от того, за сколько шагов при выводе цепочек определяется отношение предшествования. Кроме этого отношения предшествования могут устанавливаться между различными

символами. Далее будут рассмотрены грамматики простого предшествования по Вирту (отношения предшествования устанавливаются как между терминальными, так и между нетерминальными символами) и грамматики операторного предшествования Флойда (отношения предшествования устанавливаются только между терминальными символами).

6.2. Грамматики предшествования Вирта

Отношения предшествования в грамматиках Вирта определяются между любыми символами: как терминальными, так и нетерминальными.

Алгоритм свёртки терминальных цепочек по Вирту:

1. Строится для любого нетерминального символа множество левых и правых символов.
2. Определяется отношение предшествования между символами, и заносятся в матрицу (таблицу предшествования).
3. Выполняя последовательные выделения основ для свёртки, осуществляется собственно свёртка цепочек.
4. Включается семантика в алгоритм свёртки цепочки.

Множество левых и правых символов для каждого нетерминала грамматики определяется следующим образом: L – множество левых символов:
$L(u)=\{S \mid u\to\varphi \vee u\to A\varphi \wedge S\in L(A)\}$, *где* $u\in V_N$,, $S\in(V_T\cup V_N)$, $\varphi\in(V_T\cup V_N)^*$
R – множество правых символов:

$$R(u)=\begin{Bmatrix} S \mid u\to\varphi S\cup u\to\varphi A\cap S\in R(A) \\ u\in V_N, S\in(V_T\cup V_N), \varphi\in(V_T\cup V_N)^* \end{Bmatrix}$$

Отношения предшествования в грамматике по Вирту определяются следующим образом:

а) $S_1 \doteq S_2$, если существует правило вида: $u\to\varphi S_1 S_2\eta$, где u – нетерминал, $S_1,S_2\in(V_T\cup V_N)$, $\varphi,\eta\in(V_T\cup V_N)^*$.

б) $S_1 \lessdot S_2$, если существует правило вида: $u\to\varphi S_1\beta\eta$, $S_2\in L(B)$, $S_1,S_2\in(V_T\cup V_N)$, $u,B\in V_N$, $\varphi,\eta\in(V_T\cup V_N)^*$

в) $S_1 \gtrdot S_2$, если существует правило вида: $u\to\varphi A S_2\eta$, $S_1\in R(A)$, $S_1,S_2\in(V_T\cup V_N)$, $u,A\in V_N$, $\varphi,\eta\in(V_T\cup V_N)^*$ или $u\to\varphi AB\eta$, $S_1\in R(A)$, $S_2\in L(B)$, $A,B,u\in V_N$

Пример 6.1.

$S\to\perp(AB)\perp$, $A\to[A]\mid *$, $B\to[B]\mid *$

Строим множество левых и правых символов.

U	L(u)	R(u)
S	⊥	⊥
A	[*	]*
B	[+	]+

Рис. 6.1.

Матрица предшествования:

M	S	⊥	(	A	B	)	[	]	*	+
S										
⊥			≐							
(				≐			<·		<·	
A					≐		<·	≐		<·
B						≐		≐		
)		≐								
[				≐	≐		<·		<·	<·
]					·>	·>	·>	·>		·>
*					·>		·>	·>		·>
+						·>		·>		

Рис. 6.2.

Проверим принадлежность цепочки $\perp([[*]][+])\perp$ языку, выполнив свертку по Вирту:

⊥ ([[*]] [+])⊥
≐<·<·<··>

⊥ ([[A]][+])⊥
≐<·<·≐≐·>

⊥ ([A] [+])⊥
≐<·≐≐· >

⊥ (A [+])⊥
≐≐<·<··>

⊥ (A [B])⊥
≐≐<·≐≐·>

⊥ (A B)⊥ -
≐ ≐ ≐ ≐ ≐

S

Рис. 6.3.

Как видно, удалось свернуть цепочку до начального выделенного символа S, значит, цепочка принадлежит языку. Зато цепочка $\perp[+]\perp \notin$ языку.

При выполнении свертки по Вирту возможны следующие ошибки: 1)между двумя символами не существует отношения предшествования; 2)выделена основа для свертки, а подходящего правила для замены нет.

Следует заметить, что не существует общего алгоритма, позволяющего преобразовать произвольную КС-грамматику к грамматике предшествования, хотя существует правило, позволяющее выполнить это для некоторых КС-грамматик, действие которого показано на следующем примере:

$$\left.\begin{array}{l} S \to A; \\ A \to a\,A \mid a \end{array}\right\} \qquad \begin{array}{l} A \doteq; \\ A \cdot > ; \end{array} \qquad \begin{array}{l} S \to A_1; \\ A \to a\,A \mid a \\ A_1 \to A \\ A_1 \doteq; \\ A \cdot > ; \end{array}$$

Включение семантики осуществляется следующим образом: в соответствии с любым правилом вывода ставится некоторое семантическое действие, которое выполняется при замене основы для свертки на это правило.

6.3. Грамматика предшествования Флойда

Отношения предшествования в грамматиках по Флойду определяются только между терминальными символами и не допускаются в правилах вывода двух рядом стоящих нетерминалов: $u \to \varphi AB \eta$.

Алгоритм свертки по Флойду включает те же пункты, что и алгоритм свертки по Вирту.

В грамматике по Флойду множество левых и правых символов определяется следующим образом:

$$L(U) = \{S \mid (U \to S\varphi) \vee (U \to U_1\, S\varphi) \vee ((U \to U_1\varphi) \wedge (S \in L(U_1)))\}$$

$$U, U_1 \in V_N,\ S \in V_T,\ \varphi \in (V_T \cup V_N)^*$$

$$R(U) = \{S \mid (U \to \varphi S) \vee (U \to \varphi S U_1) \vee ((U \to \varphi U_1) \wedge (S \in R(U_1)))\}$$

$$U, U_1 \in V_N,\ S \in V_T,\ \varphi \in (V_T \cup V_N)^*.$$

Отношения предшествования между любыми терминальными символами S1 и S2 определяются следующим образом:

$S_1 \doteq S_2 \quad \exists (U \to \varphi S_1 S_2 \psi) \vee (U \to \varphi S_1 U_1 S_2 \psi), \quad U, U_1 \in V_N,\ S_1, S_2 \in V_T;\ \varphi, \psi \in (V_N \cup V_T)^*$

$S_1 \cdot > S_2 \quad \exists (U \to \varphi U_1 S_2 \psi) \wedge (S_1 \in R(U_1));$

$S_1 < \cdot S_2 \exists (U \to \varphi S_1 U_1 \psi) \wedge (S_2 \in L(U_1));$

Пример 6.2:

$S \to \perp B \perp$	U	$L(U)$	$R(U)$
$B \to B + T \mid T$	S	$\perp$	$\perp$
$T \to T * M \mid M$	B	$+ * a .. z ($	$+ * a .. z)$
$M \to a \mid .. \mid z$	T	$* (a .. z$	$*) a .. z$
$M \to (B)$	M	$a .. z ($	$a .. z)$

Матрица предшествования:

M S	⊥	/	+	*	a.	(	)
⊥	≐	<·	<·	<·	<·	<·	
/	·>	·>	<·	<·	<·	<·	·>
+	·>	·>	<·	<·	<·	<·	·>
*	·>	·>	·>	<·	<·	<·	·>
a..z	·>	·>	·>				·>
(		<·	<·	<·	<·	<·	≐
)	·>	·>	·>				·>

Рис. 6.4.

Выполним свертку по Флойду цепочки - $\perp a+(b*c)\perp$

$$\begin{array}{l}
<\cdot\cdot> \\
\perp M+(b*c)\perp \\
<\cdot<\cdot<\cdot> \\
\perp M+(M*C)\perp \\
<\cdot<\cdot\ <\cdot<\cdot\cdot> \\
\\
\perp M+(M*M)\perp \\
<\cdot<\cdot\ <\cdot\cdot> \\
\perp M+(T*M)\perp \\
<\cdot\ <\cdot<\cdot\ \cdot> \\
\perp M+(T)\perp \\
<\cdot<\cdot\doteq\cdot> \\
\perp M+(B)\perp \\
<\cdot<\cdot\doteq\cdot> \\
\perp M+M\perp \\
<\cdot\ \ \cdot>
\end{array}$$

$\perp T+M\perp$, $\perp T+T\perp$, $\perp B+T\perp$, $\perp B\perp$

$\doteq$

ς

Рис. 6.5.

Таким образом, можно сделать вывод, что цепочка принадлежит языку.

По сравнению с грамматикой Вирта, в грамматиках по Флойду матрица предшествования гораздо меньше, но алгоритм свертки работает гораздо дольше из-за необходимости перебора всех правил вида $A \to B$. Ошибки при свертке по Флойду возникают в следующих случаях:

1) Если между двумя символами, стоящими рядом на одном из шагов вывода цепочки, нет отношения предшествования;

2) Если для выделенной основы для свертки выполнены все переборы, а соответствующих правил для замены в грамматике нет.

Введение семантики в грамматику по Флойду осуществляется так же, как в грамматиках по Вирту.

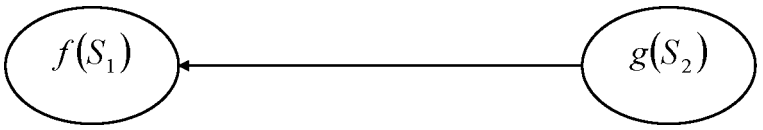
6.4 Функции предшествования

В грамматиках алгоритмических языков число терминальных и нетерминальных символов превышает 200-300, поэтому таблица предшествования занимает большой объём памяти. Для уменьшения этого объёма было предложено вместо таблицы предшествования использовать функции предшествования - числовые функции нечислового аргумента, которые определяются следующим образом:

$$f(S_1) = g(S_2): S_1 \doteq S_2$$
$$f(S_1) < g(S_2) \quad : \quad S_1 <\cdot S_2$$
$$f(S_1) > g(S_2) \quad : \quad S_1 \cdot > S_2$$

Рассмотрим *метод графов построения функции предшествования.*

В соответствии с таблицей предшествования граф строится следующим образом: для каждого символа (отношения предшествования в таблице предшествования) определяются две вершины $f(S_1)$ и $g(S_2)$, причём если для некоторых символов $S_1 \doteq S_2$, то $f(S_1)$ и $g(S_2)$ объединяются в одну вершину. Если между символами $S_1 <\cdot S_2$, то на графе они будут связаны таким образом:



,аналогично,

если $S_1 \cdot > S$, то

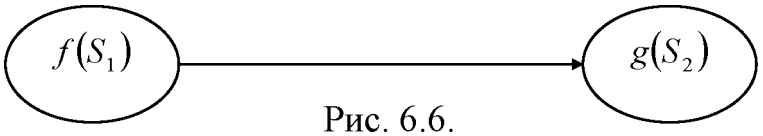


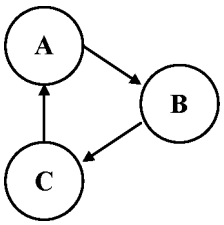
Рис. 6.6.

После построения графа выполняется следующая последовательность действий:

1) Все вершины, для которых нет потомков, помечаются индексом “1” и удаляются из рассмотрения вместе с входящими в них дугами.

2) Вершины, у которых все потомки помечены, помечаются значениями на единицу больше, чем максимальный индекс его потомков, после этого эти вершины удаляются из рассмотрения вместе с входящими в них дугами.

3) Пункт 2) выполняется до тех пор, пока все вершины не окажутся помеченными, при этом индекс вершины считается значением функции предшествования. Если в графе остаются циклы вида:



$\begin{cases} A > B \\ B > C \\ C > A \end{cases}$, то функция предшествования не существует.

Рис. 6.7.

Рассмотрим ещё один способ построения функций предшествования - метод инкрементов:

1) Все значения f и g приравниваем к единице.

2) Функции предшествования изменяются в большую сторону следующим образом:

если $S_1 <\cdot S_2$, то $g(S_2)=f(S_1)+1$

если $S_1 \cdot > S_2$, то $f(S_1)=g(S_2)+1$

если $S_1 \doteq S_2$ $\begin{cases} f(S_1)<g(S_2), \text{ то } f(S_1)=g(S_2) \\ f(S_1)>g(S_2), \text{ то } g(S_2)=f(S_1) \end{cases}$

		$\perp$	+	*	$a..z$	(	)
1 шаг	f	1	1	1	1	1	1
	g	1	1	1	1	1	1
2 шаг	f	1	3	5	5	1	5
	g	1	2	4	6	6	1
3 шаг	f	1	3	5	5	1	5
	g	1	2	4	6	6	1

Алгоритм построения функции предшествования методом инкрементов завершается успешно, если на очередном шаге значение функции предшествования совпадает со значениями функции предшествования на предыдущем шаге.

Функция предшествования не существует, если значения ее получаются больше, чем величина 2n, где n- общее число символов матрицы предшествования.

Упражнения к шестой главе

6.1. Построить отношения простого предшествования для грамматики с правилами S → aSb | c | d.

6.2. Построить отношения простого предшествования и функцию предшествования, используя граф линеаризации, для грамматики с правилами S → aSSb | c | d. Покажите этапы свертки фраз aacdbcb , ab , acb.

	1	2	3	4
1	●>		●>	● >
2	●>		●>	
3	<●	≐	<●	
4	≐		≐	

6.3.Определить, является ли грамматика логических выражений, правила которой приведены ниже, грамматикой простого предшествования Вирта. Если она таковой не является, то преобразовать ее, построить для нее отношения простого предшествования и функцию предшествования, используя метод графов. E → E or T | T

T → T and M | M

M → a | (E) | not M

6.4. Постройте отношения простого предшествования для грамматики с правилами S → 0S11 | 011 и терминалами { 0, 1 }. Если данная грамматика не является грамматикой простого предшествования, то преобразуйте ее к такой грамматике.

6.5. Постройте отношения операторного предшествования и функцию предшествования, используя методы графов и инкрементов, для грамматики из упражнения 6.4.

6.6. Преобразуйте грамматику $S \to E$

$E \to T \mid TBE \mid (E)$

$T \to a \mid b \mid \ldots \mid z$

$B \to + \mid - \mid * \mid /$ к грамматике простого предшествования Вирта. Покажите этапы свертки фразы a*(b+c).

6.7. Написать матрицу предшествования Флойда для грамматики:

$S \to \perp AB \perp$

$A \to aA$

$A \to a$

$B \to (B)$

$B \to C$

$C \to 0 \mid \ldots \mid 9 \mid 0C \mid \ldots \mid 9C$

и проанализировать цепочку: 1) $\perp$aa(((039))) $\perp$ 2) $\perp$aa(497)) $\perp$

Глава 7. Введение в семантику

Обычно в компиляторах и интерпретаторах каждому правилу грамматики, каждой альтернативе любого нетерминала ставятся в соответствие семантические подпрограммы. Эти подпрограммы выполняются при синтаксических редукциях по заданным правилам грамматики в восходящем разборе или отождествлении фрагмента входной цепочки с некоторой альтернативой продукции при разборе нисходящем.

В тех случаях, когда исходный язык программирования достаточно сложен или к компилятору предъявляются повышенные требования (например, необходима машинно–независимая оптимизация исходной программы с целью получения более эффективного объектного кода), первоначально исходная программа переводится в некоторую внутреннюю форму, более удобную для простой машинной обработки. В большинстве внутренних представлений операторы располагаются в том порядке, в котором они должны выполняться, что существенно облегчает последующий анализ, интерпретацию или генерацию объектного кода.

Все внутренние представления программы обычно содержат элементы двух типов: операторы и операнды. Различия представлений состоят лишь в том, как эти элементы объединяются между собой. В дальнейшем мы будем использовать такие традиционные операторы, как +, –, /, MOD, DIV, *, AND, OR, >, <, = и т. п., а также БП (Безусловный Переход) и УПЛ (Условный Переход по Лжи), точнее условный переход в том случае, когда значение операнда (логического выражения) – ложь (FALSE, 0). Внутри компилятора, конечно же, все они представляются соответствующими лексемами или целочисленными кодами.

Операнды, с которыми мы будем иметь дело, – это простые идентификаторы (имена переменных, процедур и т.п.), константы, временные переменные, генерируемые самим компилятором, и переменные с индексами.

7.1. Польская инверсная запись

Вместо традиционного инфиксного (скобочного) представления арифметических и логических выражений в различных вычислителях часто используется польская инверсная запись (ПОЛИЗ), которая просто и точно указывает порядок выполнения операций без использования скобок. В этой записи, впервые примененной польским математиком Я.Лукашевичем, операторы располагаются непосредственно за операндами над которыми они выполняются в порядке их выполнения. Поэтому иногда ПОЛИЗ называют суффиксной или постфиксной записью. Например, A+B записывается как AB+, A*B+C – как AB*C+, A*(B+C/D) – как ABCD/+*, а A*B+C*D – как AB*CD*+.

Остановимся на простейших правилах, которые позволяют переводить в ПОЛИЗ вручную:

1). Идентификаторы и константы в ПОЛИЗе следуют в том же порядке, что и в инфиксной записи.

2). Операторы в ПОЛИЗе следуют в том порядке, в каком они должны вычисляться (слева направо).

3). Операторы располагаются непосредственно за своими операндами.

Таким образом, мы могли бы записать следующие синтаксические правила:

<операнд>::=идентификатор | константа | <операнд><операнд><оператор>

<оператор>::=+ | – | * | / | …

Унарный минус и другие унарные операторы можно представить двумя способами: либо записывать их бинарными операторами, то есть вместо –B писать 0–B, либо для унарного минуса можно ввести новый специальный символ, например @, и использовать еще одно синтаксическое правило <:операнд>::=<операнд>@. Таким образом выражение A+(–B+C*D) мы могли бы записать AB@CD*++.

С равным успехом мы могли бы ввести префиксную запись, где операторы стоят перед операндами. Таким образом, арифметическое выражение, а далее мы покажем, что не только его, но и любую управляющую конструкцию, можно представить в трех формах записи: префиксной, инфиксной (обычная запись, где операторы располагаются между операндами, а круглые скобки позволяют изменять приоритет операций) и постфиксной. Человек традиционно использует инфиксную запись, тогда как для автоматического вычисления выражений самым удобным способом представления является постфиксная запись или ПОЛИЗ.

ПОЛИЗ расширяется достаточно просто. Нужно только придерживаться правила, что за операндами следует соответствующий им оператор. Так присваивание <переменная>:=<выражение> в ПОЛИЗе примет вид

<переменная><выражение>:=. Например, присваивание A:=B*C+D*100 запишется в ПОЛИЗе как ABC*D100*+:=.

Индексированную переменную в ПОЛИЗе, а точнее вычисление ее адреса можно представить в виде:

идентификатор<индексные выражения>константа[,

где [– обозначает знак операции вычисления индекса, идентификатор – имя (базовый адрес) индексированной переменной, а константа – количество индексов (мерность массива). Так переменную A[i,j+k] можно представить в виде Aijk+2[.

Условный оператор

IF <выр> THEN <инстр $_1$> ELSE <инстр $_2$>

в ПОЛИЗе будет иметь вид:

<выр> < m > УПЛ <инстр $_1$> < n > БП <инстр $_2$> , где

- <выр> – логическое выражение (условие), которое может принимать значения – 0 (FALSE, ложь) или 1 (TRUE, истина);
- < m > – номер (место, позиция, индекс) символа ПОЛИЗа, с которого начинается <инстр $_2$>;
- УПЛ (Условный Переход по Лжи) – оператор с двумя операндами <выр> и < m >, смысл которого состоит в том, что он изменяет традиционный порядок вычислений и осуществляет переход на символ строки ПОЛИЗа с номером < m >, если (и только если) <выр> – ложно (равно 0);
- < n > – номер символа, следующего за <инстр $_2$>;
- БП (Безусловный Переход) – оператор с одним операндом < n >, который также изменяет порядок вычислений по ПОЛИЗу и осуществляет переход на символ с номером < n >.

Операторы условного и безусловного перехода, как в свое время было показано Дейкстрой, составляют основу внутреннего представления любой структурной управляющей конструкции (циклов типа FOR, WHILE, REPEAT, оператора выбора и т.п.). В рассматриваемых нами примерах потребуется только один условный оператор – УПЛ, хотя никто не мешает определить целую группу таких операторов (смотри, например, мнемонику команды условного перехода языка ассемблера для ПЭВМ с процессором Intel 8086).

Пример 7.1.

В заключении раздела рассмотрим пример перевода в ПОЛИЗ фрагмента программы, включающего условный оператор:

IF (x<y) AND (a< >0) THEN b:=a+b*c ELSE b:=(a+b)*c; x:=a*b+c*d;

Ниже приведена строка ПОЛИЗа для этого оператора, где над символами указаны номера их позиций в полученной строке:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
x	y	<	a	0	<	>	AND	19	УПЛ	b	a	b	c	*	+	:=	26	БП	b	a	b	+	c	*	:=	x	a	b	*	c	d	*	+	:=

7.2. Интерпретация ПОЛИЗа

С помощью стека арифметическое выражение в ПОЛИЗе может быть вычислено за один просмотр слева направо. В стеке будут находиться все

операнды, которые встретились при просмотре строки ПОЛИЗа или получились в результате выполнения некоторых операций, но еще не использовались в вычислениях. Алгоритм обработки строки ПОЛИЗа состоит в следующем:

1). Если сканируемый символ идентификатор или константа, то соответствующее им значение заносится в стек и осуществляется переход к следующему символу строки ПОЛИЗа. Это соответствует использованию правила <:операнд>::=идентификатор | константа

2). Если сканируемый символ унарный оператор, то он применяется к верхнему операнду в стеке, который затем заменяется на полученный результат. С точки зрения семантики это соответствует применению правила
<:операнд>::= <операнд><оператор>.

3). Если сканируемый символ бинарный арифметический или логический оператор, то он применяется к двум верхним операндам в стеке, и затем они заменяются на полученный результат. Это соответствует использованию правила <:операнд>::= <операнд><операнд><оператор>.

Одно из исключений – бинарный оператор присваивания, который в ПОЛИЗе имеет вид <пер><выр>:=. После его выполнения и <пер>, и <выр> должны быть исключены из стека, так как оператор ':=' не имеет результирующего значения. При этом в стеке должно находится не значение <пер>, а ее адрес, так как значение <выр> должно сохраняться по адресу <пер>.

4). Бинарный оператор УПЛ, операндами которого является уже вычисленное значение логического выражения <выр> и номер символа строки ПОЛИЗа – <m>, также удаляет оба операнда из стека и не формирует в нем результата. Его действие состоит в том, что он анализирует значение выражения, и если оно равно 1 (истинно), то следующим сканируется символ, расположенный сразу за УПЛ. Если же значение выражения – 0 (ложно), то мы перемещаемся по строке ПОЛИЗа к символу, позиция которого указана в операнде <m>.

5). Унарный оператор БП, удаляя свой параметр, – номер символа < n > из стека, просто переводит сканирование к указанной позиции ПОЛИЗа.

Пример 7.2.

На рис. 7.1 показан пример интерпретации строки ПОЛИЗа AB@CD*++ , полученной из инфиксного выражения A+(−B+C*D). Значения в стеке отделены друг от друга вертикальной чертой.

Анализируемая строка	Номер правила	Старое состояние стека	Новое состояние стека	Новое состояние стека в числовой форме
▽AB@CD*++	1		A	1
A▽B@CD*++	1	A	A | B	1 | 2
AB▽@CD*++	2	A | B	A | –B	1 | –2
AB@▽CD*++	1	A | –B	A | –B | C	1 | –2 | 3
AB@C▽D*++	1	A | –B | C	A | –B | C | D	1 | –2 | 3 | 4
AB@CD▽*++	3	A | –B | C | D	A | –B | C*D	1 | -2 | 12
AB@CD*▽++	3	A | –B | C*D	A | –B+C*D	1 | 10
AB@CD*++▽	3	A | –B+C*D	A+(–B+C*D)	11

Рис.7.1.

7.3. Генерирование команд по ПОЛИЗу

При генерировании команд по ПОЛИЗу также используется стек, в котором вместо значений сохраняются адреса или символические имена операндов, которые в процессе работы алгоритма заменяются на имена ячеек временной памяти для результатов промежуточных вычислений. Основу формирования команд составляют кодовые продукции – набор машинных команд, который соответствует отдельному оператору ПОЛИЗа.

Алгоритм генерации команд состоит в том, что строка ПОЛИЗа просматривается один раз слева направо и при встрече операндами (идентификаторами или константами) их имена (символические значения, адреса) заносятся в стек. При встрече с оператором из таблицы выбирается соответствующая ему кодовая продукция. В нее подставляются имена (адреса) операндов, извлеченные из стека, а так же сформированное имя (адрес) результата. Последнее имя замещает операнды данного оператора в стеке, а сформированная продукция помещается в выходной файл. Ниже все примеры даются с использованием языка ассемблера для IBM PC (процессор Intel 8x86). Если Вы еще не знакомы с этим языком, то надеемся, что комментарии к командам позволят понять их смысл.

Пример 7.3.

На рис. 7.2 показан пример генерирования команд для уже известной нам строки ПОЛИЗа AB@CD*++.

Анализируемая строка	Новое состояние стека	Сгенерированные команды
▽ **AB@CD*++**	**A**	
▽ **AB@CD*++**	**A | B**	
▽ **AB@CD*++**	**A | M_1**	**MOV AX, B ; пересылка B в AX** **NEG AX ; вычисление –B в AX** **MOV M1, AX ; результат в M_1**
▽ **AB@CD*++**	**A | M_1 | C**	
▽ **AB@CD*++**	**A | M_1 | C | D**	
▽ **AB@CD*++**	**A | M_1 | M_2**	**MOV AX, C ; пересылка C в AX** **MUL D ; вычисление C* D** **MOV M2, AX ; результат в M_2**
▽ **AB@CD*++**	**A | M_3**	**MOV AX, M1 ; пересылка M_1 (–B) в AX** **ADD AX, M2 ; вычисление M_1+M_2 (–B+C*D)** **MOV M3, AX ; результат в M_3**
AB@CD*++▽	**M_4**	**MOV AX, A ; пересылка A в AX** **ADD AX, M3 ; вычисление A+M_3 (A+(–B+C*D))** **MOV M4, AX ; результат в M_4**

Рис. 7.2.

Полученная в примере программа далека от оптимальной. Если поменять местами операнды в симметричных операциях (здесь для умножения и сложения), то программа будет более эффективной.

ПОЛИЗ является прекрасной иллюстрацией для внутренней (промежуточной) формы представления исходной программы. Алгоритмы интерпретации ПОЛИЗа и генерации команд по ПОЛИЗу предельно просты, но с точки зрения машинно–независимой оптимизации эта форма не совсем удобна. Идеальными здесь являются представления бинарных операций в виде тетрад (четверок):

<оператор>, <операнд $_1$>, < операнд $_2$>, <результат> ,

где <операнд $_1$> и <операнд $_2$> специфицируют аргументы, а <результат> – результат выполнения оператора над аргументами. Таким образом, A*B мы могли бы представить, как

*, A, B, M ,

где M – некоторая временная переменная, которой присваивается результат вычисления A*B. Аналогично A*B+C*D представляется в виде следующей последовательности тетрад:

*, A, B, M1

*, C, D, M2

+, M1, M2, M3

Важно отметить, что в отличии от обычной инфиксной записи тетрады располагаются в том порядке, в котором они должны выполняться. Унарные операторы также оформляются в виде тетрад, но <операнд $_2$> остается в них пустым. Так, вместо –A появится тетрада “–, A, , M”, что означает “присвоить M

значение –А”. Унарный минус, также как и в ПОЛИЗе, мы могли бы заменить другим символом (кодом), чтобы отличать его от бинарного.

Кроме традиционной арифметики в виде тетрад можно представить и любой другой оператор, имевший место в польской записи. Оператор присваивания A:=B представим в виде тетрады “ :=, B, , A ”, а оператор УПЛ запишется в виде тетрады “ УПЛ, <выр>, <m> , ”, где <m> – номер тетрады, на которую будет осуществляться переход, если значение логического выражения (<выр>) – равно нулю (ложно).

К недостаткам тетрад можно отнести большое количество временных переменных, требующих описания. Эти проблемы полностью отпадают при использовании триад (троек), которые имеют следующую форму:
<оператор> <операнд $_1$>, < операнд $_2$>

В триаде нет поля для результата. Если позднее какой–либо операнд окажется результатом данной операции, то он будет непосредственно на нее ссылаться (на операцию, а точнее соответствующую триаду). Например, выражение A+B*C будет представлено следующим образом:
(1) * B, C
(2) + A, (1)
Здесь (1) – ссылка на результат первой триады, а не константа, равная 1. Выражение 1+B*C будет записываться так:
(1) * B, C
(2) + 1, (1)

Конечно, в компиляторе мы должны отмечать этот тип операнда, используя новый код в первом байте его представления. Триада занимает меньше места, чем тетрада, но следует помнить, что при работе с триадами нам придется хранить описания результатов, значения которых в дальнейшем еще потребуются.

Достоинства использования тетрад и триад с точки зрения машинно–независимой оптимизации по сравнению с ПОЛИЗом очевидны. Представляя их в виде таблицы (односвязного или двухсвязного списка), тетрады или триады можно легко переставлять или удалять “лишние”.

7.4. Алгоритм Замельсона и Бауэра перевода выражений в ПОЛИЗ

Арифметические выражения чаще всего встречались при практическом программировании, особенно на ранних стадиях использования вычислительной техники. Поэтому для них немецкими математиками К. Замельсоном и Ф. Бауэром был предложен достаточно простой метод перевода инфиксных арифметических выражений в ПОЛИЗ с одновременным контролем отдельных ошибок. Предложенный ими метод не использует грамматик в явном виде но в основе приоритетов операций о которых речь пойдет ниже лежат традиционные функции предшествования.

Метод Замельсона–Бауэра для перевода инфиксных арифметических выражений в ПОЛИЗ:

Вход. Строка, содержащая инфиксное арифметическое выражение.

Выход. Польская инверсная запись исходного выражения.

Метод. Суть метода состоит в том, что для каждого знака–разделителя (операции, скобки и т.п.) вводят два числа: сравнительный приоритет – P_C и магазинный приоритет – P_M. Исходное выражение просматривается один раз слева направо. При этом:

Идентификаторы и константы переписываются в выходную строку ПОЛИЗа.

При обнаружении разделителя его сравнительный приоритет P_C сравнивается с магазинным приоритетом P_M разделителя из вершины магазина операций. Если $P_C > P_M$, то разделитель входной строки помещается в магазин (разделитель из исходной строки поступает в магазин и в том случае, когда магазин пуст). Если $P_C \le P_M$, то символ извлекается из магазина и записывается в выходную строку ПОЛИЗа. Далее повторяется пункт (2) все для того же входного символа.

Открывающая скобка – ‘(’ имеет самый высокий сравнительный приоритет и поэтому всегда поступает в магазин. Закрывающая скобка – ‘)’в ПОЛИЗ и магазин не записывается и поэтому магазинный приоритет для нее не важен. По закрывающей скобке входной строки из магазина извлекаются все операции и переписываются в строку ПОЛИЗа вплоть до первой открывающей скобки в магазине. Открывающая скобка также извлекается из магазина, но, как и закрывающая, в ПОЛИЗ не переписывается. После этого осуществляется переход к следующему символу входной строки. Ясно, что если для закрывающей скобки входной строки в магазине открывающая скобка не будет обнаружена, то это послужит сигналом синтаксической ошибки.

Символ	**(**	**)**	**+ –**	*** /**
P_C	**100**	**0**	**2**	**3**
P_M	**0**		**2**	**3**

Рис.7.3.

По окончании входной строки содержимое магазина переписывается в строку ПОЛИЗа. (Если при этом в магазине останутся открывающие скобки, то это также является признаком ошибки в записи исходного выражения). □

На рис. 7.3 представлена таблица приоритетов операций и скобок для упрощенных арифметических выражений, а на рис. 7.4 разобран по шагам пример перевода в ПОЛИЗ инфиксного выражения по методу Замельсона–Бауэра. (На рис. 7.4 вершина магазина операций расположена справа).

Предложенный метод можно модифицировать таким образом, что он позволит обрабатывать и операции сравнения, логические операции, операторы присваивания, управляющие конструкции типа IF–THEN–ELSE, WHILE–DO и т. п. Таблица приоритетов для этого случая представлена на рис. 7.5. То, что в ней приоритеты операций изменены, по сравнению с рис. 7.3, роли не играет. Важны отношения между значениями приоритетов, а не сами значения. Обработка скобок, к которым здесь можно отнести и операцию присваивания – ‘:=’, и знак конца оператора – ‘;’, и элементы управляющих конструкций (IF,

THEN, ELSE, WHILE и т.п.) выполняется по особым алгоритмам, часть из которых была предложена в работе Л.Ф. Штернберга [17].

Результирующая строка ПОЛИЗа	Магазин	Необработанная часть входной строки
		a+b*(c–d/(10+x)+y)–e
a	+	b*(c–d/(10+x)+y)–e
a b	+ *	(c–d/(10+x)+y)–e
a b	+ * (	c–d/(10+x)+y)–e
a b c	+ * (–	d/(10+x)+y)–e
a b c d	+ * (– /	(10+x)+y)–e
a b c d	+ * (– / (	10+x)+y)–e
a b c d 10	+ * (– / (+	x)+y)–e
a b c d 10 x	+ * (– / (+	)+y)–e
a b c d 10 x +	+ * (– /	+y)–e
a b c d 10 x + /	+ * (–	+y)–e
a b c d 10 x + / –	+ * (	+y)–e
a b c d 10 x + / –	+ * (+	y)–e
a b c d 10 x + / – y	+ * (+	)–e
a b c d 10 x + / – y +	+ *	–e
a b c d 10 x + / – y + *	+	–e
a b c d 10 x + / – y + * +		–e
a b c d 10 x + / – y + * +	–	e
a b c d 10 x + / – y + * + e	–	
a b c d 10 x + / – y + * + e –		

Рис.7.4.

При дальнейших рассуждениях, для того чтобы упростить изложение, будем считать, что любая управляющая конструкция, будь то оператор IF–THEN–ELSE или WHILE–DO, не содержит BEGIN, но всегда завершается терминалом END, независимо от количества операторов в отдельной ветви или теле цикла. (Смотрите, например язык МОДУЛА–2).

Символ	([IF := WHILE	THEN DO	ELSE	)] ; END	OR	AND	NOT	< > = <> <= >=	+ –	* /
P_C	100	1	2	1	5	6	7	8	9	10
P_M	0	0	2		5	6	7	8	9	10

Рис.7.5.

Операция присваивания ‘:=’ играет роль открывающей скобки для символа ‘;’ (или управляющей конструкции, типа ELSE или END, если символ ‘;’ перед ними необязателен). По ‘;’ из магазина выталкивается все вплоть до операции присваивания и она сама извлекается из магазина и переписывается

в ПОЛИЗ. При этом, если символу присваивания в магазине предшествуют “открывающие скобки” иного рода, то это является признаком ошибки.

7.5. Атрибутные грамматики

Способ описания грамматик с помощью порождающих правил не единственный. Порождающие грамматики, рассмотренные ранее, не учитывают в формальной записи грамматик семантические правила. Семантические действия в таких грамматиках вводятся в виде процедур при построении анализаторов. Примерами грамматик, учитывающих семантику в правилах вывода, являются:

двухуровневые грамматики W-грамматики(с их помощью описан язык ALGOL-68),

венский метаязык (описан PL/1),

атрибутные.

Ниже рассмотрен один из перечисленных видов грамматик, а именно – атрибутные грамматики [12]. *Атрибут*- это свойство объекта; под атрибутом в грамматиках понимаются значение или семантический смысл (значимость) объекта. *Атрибутная грамматика* – это КС-грамматика, с узлами дерева вывода которой связаны атрибуты (семантические правила). КС-правилам сопоставляются правила вычисления атрибутов. Правило вычисления значений атрибутов, соответствующее данному КС-правилу, применяется для всех вхождений этого правила в дерево вывода.

Атрибуты могут быть двух видов - *синтезированные и унаследованные*.

Синтезированные атрибуты вычисляются с учетом значений атрибутов узлов потомков.

Унаследованные атрибуты - это атрибуты, значение которых вычисляется с учетом значений атрибутов его предков.

Формально **атрибутная грамматика** – это пятёрка объектов:

$G_A = < V_T, V_N, S, R, A(x) >$, где

V_T - множество терминальных символов;

V_N - множество нетерминальных символов;

S - начальный выделенный символ;

R - это правила вывода.

$A \to \zeta$,где $\zeta \in (V_T \cup V_N)^*$

$A(x)$- это множество атрибутов

$A(x) = A_y(x) \cup A_c(x)$, где

$A_y(x)$- множество унаследованных атрибутов;

$A_c(x)$- множество синтезированных атрибутов.

$A_y(x) \cup A_c(x) = \varnothing$

$A_y(S) = \varnothing$

$A_c(a) = \varnothing$

$\forall\ a \in V_T$

Унаследованные атрибуты левой части КС-правила и синтезированные атрибуты правой части получают значения, переданные от окружающих ветвлений дерева вывода. Правила вычисления значений атрибутов, связанные с каким-либо КС-правилом, определяют метод вычисления значений других атрибутов, а именно – унаследованных атрибутов правой части и синтезированных атрибутов левой части. Значения этих атрибутов передаются окружающим узлам. Или иначе можно сказать, что синтезированные атрибуты некоторого узла содержат информацию, которая синтезируется из поддерева данного узла и передаётся вверх к корню дерева вывода, а в унаследованных атрибутах хранится информация, передаваемая вниз, от корня дерева к его листьям. Унаследованные атрибуты характеризуют контекст, в котором находится его узел и его поддерево. Правила вычисления значений атрибутов записываются разными способами. Аппарат атрибутных грамматик не является законченным методом формального описания языка. Для того, чтобы им можно было воспользоваться, его необходимо дополнить методом записи правил вычисления значений атрибутов.

Один из возможных методов записи правил вычисления значений атрибутов рассмотрим на примере атрибутной грамматики чисел в двоичной системе записи.

Контекстно-свободная грамматика имеет вид:

$S \to L.L \mid L,\ L \to L\ B \mid B,\ B \to 0 \mid 1$

Здесь нетерминал S используется для вывода числа в двоичной системе записи, нетерминал L – для вывода списка бит (0 и 1), нетерминал B – позволяет вывести один бит (0 или 1). В эту грамматику включим атрибуты (семантические правила), позволяющие в процессе анализа правильной цепочки получить значение двоичного числа в десятичной системе счисления. Для этого каждому нетерминалу припишем атрибуты следующим образом:

1°. Каждое число (нетерминал S) имеет один атрибут $\delta(S)$, равный значению числа в десятичной системе счисления.

2°. Каждый список битов (нетерминал L) имеет два атрибута - значение последнего бита $\delta(L)$ и длину $l(L)$.

3°. Каждый бит (нетерминал B) имеет один атрибут- значение $\delta(B)$.

В общем случае каждому нетерминалу можно приписывать любое число атрибутов. Описанные выше семантические правила относятся к синтезированным атрибутам, така как значения атрибутов всех нетерминалов определяются через значения атрибутов потомков.

С учетом этих правил построим атрибутную грамматику:

$S \to L_1 \cdot L_2 \quad \delta(S) = \delta(L_1) + \delta(L_2)/2^{l(L_2)}$

$S \to L \quad \delta(S) = \delta(L)$

$L_1 \to L_2 B \quad l(L_1) = l(L_2) + 1;\ \delta(L_1) = 2 * \delta(L_2) + \delta(B)$

$\mathrm{L} \to \mathrm{B} \quad \delta(L) = \delta(B),\ l(L) = 1$

$\mathrm{B} \to 0 \quad \delta(B) = 0$

$\mathrm{B} \to 1 \quad \delta(B) = 1$

Здесь индексы у нетерминалов L1 и L2 использованы для того, чтобы различить вхождения одноименных нетерминалов в правой и левой частях правил вывода. Ниже построено дерево вывода для числа 101.01, а в скобках получено значение числа в десятичной системе счисления.

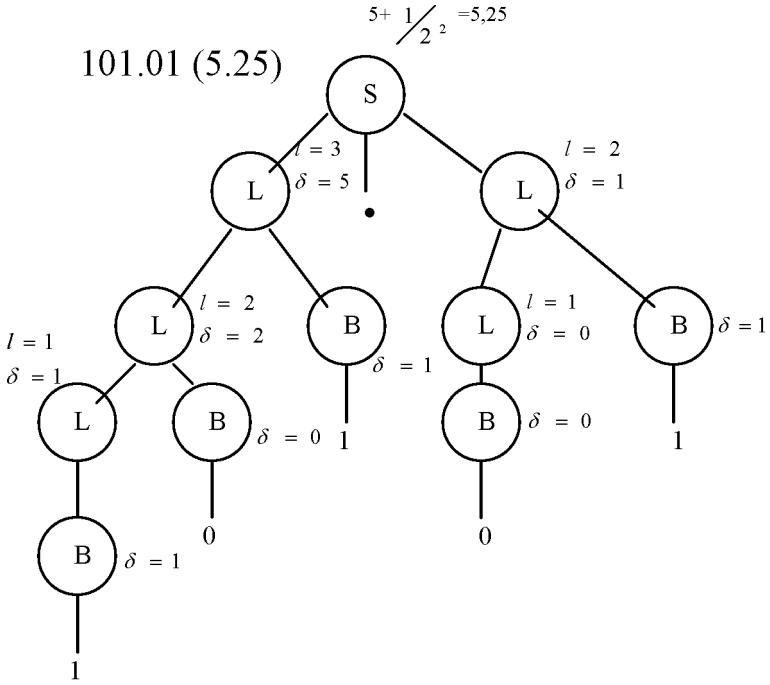


Рисунок 7.6. Дерево вывода для числа 101.01

Упражнения к седьмой главе

7.1. Перевести в ПОЛИЗ, используя метод Замельсона – Бауэра, следующий фрагмент программы:

```
i:=a+b*c;
if ( (i > 0) and (y < x+10) ) then begin
  x := b+10*a*y/(b-x); j:=15;
  while (j < 20) do begin y:=(y+2)*a; j:=j+1; end;
  i:=(i −a)*b/c;
end; a:=a −b/c −d;
```

Проинтерпретируйте полученную строку ПОЛИЗа и сгенерируйте по ней эквивалентный набор команд на языке ассемблера.

7.2. Перевести в тетрады, используя метод Замельсона – Бауэра, следующий фрагмент программы:

```
if ( (x <= 100) or (y < > x+10) and (y > b) then
  x := −(a+b DIV 10)*y MOD b-x;
else begin y:=a−b*c+d; if y < a then y:=y+2*a; x :=y+a−b; end;
a:=a*b+c*d;
```

Сгенерируйте по полученным тетрадам эквивалентный набор команд на языке ассемблера.

Глава 8. Основные фазы компиляции

Транслятор – это программа перевода текста (программы) с одного языка (исходного) на другой (объектный). Трансляторы различают компилирующего и интерпретирующего типов. Компилятор переводит всю программу, затем её выполняет. Интерпретатор переводит по отдельным операторам программу и сразу каждый из операторов исполняет.

Компилятор должен выполнить анализ исходной программы, а затем синтез объектной. Сначала исходная программа разлагается на составные части; затем из них строятся фрагменты эквивалентной объектной программы. Для этого на этапе анализа компилятор использует и строит целый ряд таблиц, структур данных, которые затем используются как при анализе, так и синтезе. Анализ процессов компиляции позволяет выделить 7 различных логических задач – фаз компиляции. В практических реализациях грани между этими фазами размыты, часть из них может отсутствовать, совмещаться одна с другой.

В этой главе мы лишь кратко остановимся на основных фазах и базах данных компилятора.

1). Лексический анализ – распознавание базовых элементов языка, перевод исходной программы в таблицу стандартных символов (лексем), которые в отличие от элементов исходной программы имеют постоянную длину, что делает последующие фазы компиляции более простыми. Лексический анализатор или сканер группирует определенные терминальные символы исходной программы в единые синтаксические объекты – лексемы. Какие объекты считать лексемами, зависит от определения языка программирования. Лексема – это пара вида: тип лексемы, некоторые данные. Первой компонентой пары является синтаксическая категория, такая как “константа”, “идентификатор” или “терминал” (ключевое слово языка или специальный символ: знак операции, разделитель и т.п.), а вторая – указатель: в ней указывается номер элемента таблицы, хранящий подробную информацию об этой конкретной лексеме. Входной информацией сканера является исходная программа и таблица терминалов языка, выходом – цепочка лексем, таблицы идентификаторов и констант.

2). Синтаксический анализ или разбор – использует только первые компоненты лексем – их типы. Информация о каждой лексеме (вторая компонента) используется на более поздних этапах процесса трансляции. Синтаксический анализ призван рассматривать базовые конструкции языка, исследовать цепочку лексем и устанавливать, удовлетворяет ли она структурным условиям, явно сформулированным в определении синтаксиса языка. Основа синтаксического анализа – синтаксические правила или грамматика языка. По предложенной грамматике можно автоматически построить синтаксический анализатор, который будет проверять, имеет ли исходная программа синтаксическую структуру, определяемую правилами грамматики. В предыдущих разделах изложены несколько методов разбора и алгоритмов построения синтаксических анализаторов по заданной грамматике.

3). Семантический анализ – определение смыслового значения базовых синтаксических конструкций. Этот процесс синтаксически управляем. То есть фазы 2 и 3 тесно связаны (объединены).

Как только синтаксический анализатор узнает конструкцию исходного языка, он вызывает соответствующую семантическую процедуру или программу, которая контролирует конструкцию с точки зрения семантики и запоминает информацию о конструкции в таблицах идентификаторов и констант либо в промежуточной (внутренней) форме исходной программы. Например, когда распознается описание переменных или констант, семантическая программа проверяет идентификаторы, указанные в этом описании, чтобы убедиться в том, что они не были описаны дважды и заносит их атрибуты или значения в соответствующие таблицы.

Когда встречается оператор присваивания вида

<переменная>:=<выражение>

семантическая программа проверяет переменную и выражение на соответствие типов, а затем заносит информацию об инструкции присваивания во внутреннюю форму программы (ВФП).

Таким образом, анализаторы 2 и 3 выполняют сложную и наиболее существенную работу по расчленению исходной программы на составные части, формированию ее внутреннего представления с занесением информации в таблицы идентификаторов и констант, осуществляют полный синтаксический и семантический контроль программы, включая действия по локализации, идентификации и нейтрализации ошибок.

4). Машинно–независимая оптимизация ВФП – вынесение общих подвыражений, вычисления над константами, оптимизация переходов в сложных условных операторах, вынесение инвариантных вычислений за цикл и т.п.

5). Распределение памяти – модификация таблиц идентификаторов и констант. Определение адресов идентификаторов и констант. Вставки в ВФП, для генерации и распределения динамической памяти. Выделение временной памяти, выравнивание и т.п.

6). Генерация кода и машинно–зависимая оптимизация. С каждой операцией из ВФП связана кодовая продукция, которая и выносится в код сборки. Оптимизация же проводится с целью более эффективного использования регистров ЭВМ, удаление “лишних” команд, связанных с сохранением и загрузкой промежуточных данных на этапе вычислений и т.п.

7). Сборка и выдача – разрешение символических адресов (трансляция с языка ассемблера) и формирование объектного модуля (машинного кода и информации для компоновщика и загрузчика).

Сразу же отметим, что фазы 1 – 4 машинно–независимы и определяются только исходным языком, а фазы 5 – 7 машинно–зависимы и не зависят от исходного языка.

а). Исходная программа – это программа на исходном языке программирования, например, С или Паскаль.

б). Таблица терминальных символов – постоянная таблица, в которой записаны все ключевые слова (IF, THEN, ELSE, WHILE и т.п.) и специальные символы языка (пробел, ‘,’, ‘;’, ‘*’, ‘+’ и т.п.) в символьной форме. На них ссылаются стандартные символы – лексемы программы.

в) Таблица (строка) лексем (стандартных символов) – состоит из полного или частичного списка лексических единиц, расположенных в том порядке, в каком они встречаются в программе (например, TRM(1÷n), IDN(1÷m), CON(1÷k)). Они создаются при лексическом анализе и используются на этапах синтаксического и семантического анализа.

г). Таблица идентификаторов – содержит информацию обо всех переменных программы, в том числе и временных переменных, хранящих промежуточные результаты вычислений, и информацию, необходимую для ссылок (адресации) и отведения памяти. Создается на фазе лексического анализа (1) (на элементы таблицы идентификаторов ссылаются лексемы), модифицируется на фазах семантического анализа (3) и распределения памяти (5), используется также на фазах генерации кода (6), сборки и выдачи (7).

д). Таблица констант – содержит все константы исходной программы и дополнительную информацию о них. Создается на фазе лексического анализа (1) (на нее ссылаются лексемы), модифицируется на фазе семантического анализа и распределения памяти, используется при генерации кода, сборке и выдаче.

е). Правила грамматики (продукции) – могут представляться и неявно в самом теле программы синтаксического анализа. Они обеспечивают, в соответствии с заданным алгоритмом, автоматический анализ синтаксиса исходной программы. Зачастую, в грамматике содержится информация и о семантических действиях (транслирующие и атрибутные грамматики), которые надо предпринимать в процессе обнаружения тех или иных языковых конструкций. ж). Внутренняя форма программы (ВФП) – форма обеспечивающая однопроходную генерацию кодов (в компиляторе) или интерпретацию (выполнение интерпретатором). Пример ВФП – ПОЛИЗ (польская инверсная запись) – где арифметические выражения, да и вся программа представляется не в традиционной инфиксной форме, а в постфиксной или суффиксной бесскобочной формах. В ПОЛИЗе операции располагаются за операндами, над которыми они выполняются в порядке их выполнения.

Например, оператор

a:=b+c*d/(b–c)–10; в ПОЛИЗе примет вид abcd*bc–/+10–:=

Еще чаще в компиляторах в качестве ВФП используется матрица тетрад, где выражение представляется в форме тетрад (оператор, операнд, операнд, результат) в порядке их выполнения. Например, присваивание a:=b+c*d будет представлено как

```
*  C ,D , , M1  ,   M1
   B  , M1, M2
   M2,    , A
+  ,  B  ,  M1,  M2
:= ,  M2,      ,  A
```

где M1 и M2 временные переменные, образованные компилятором. (При работе компилятора, операндами в приведенных примерах будут не сами символические имена и значения, а лексемы – ссылки на таблицы, где они были описаны.) ВФП создается на фазе семантического анализа, модифицируется (оптимизируется) на фазе машинно–независимой оптимизации и используется при генерации кода.

з). Кодовые продукции – постоянная таблица, имеющая отдельные элементы, определяющие код для каждой возможной операции ПОЛИЗа или матрицы тетрад (т.е. ВФП). Например, тетрада +, операнд_1, операнд_2, результат или более конкретно +, A, B, M10 может быть представлена следующей кодовой продукцией:

```
        MOV   ax, A
    ADD   ax, B
    MOV   ax, M10
```

а тетрада :=, операнд_1,, результат (:=, M20,,ABC) – продукцией

```
        MOV   ax, M20
    MOV   ABC, ax
```

Таблица кодовых продукций используется на фазе генерации кода.

и). Код сборки – версия программы на языке сборки (аналог языка ассемблера). Создается на фазе генерации кода и используется фазой сборки.

к). Перемещаемый объектный модуль – результат фазы сборки и всей трансляции в целом. Является входной информацией для компоновщика или загрузчика.

Более подробно описание каждой фазы компиляции рассмотрено, например в [7,11].

Заключение

Завершая курс «Основы теории формальных грамматик» следует отметить, что его рамки не позволили рассмотреть и малой толики того объема знаний, который накоплен в данной области. Здесь приведены лишь наиболее важные фрагменты стройной теории компиляции. Заинтересованный читатель откроет для себя массу полезного, если познакомится с работами, приведенными в списке литературы. Обсуждаемые там методы и алгоритмы играют большую роль не только в компиляции, - это часть общей культуры программирования и искусственного интеллекта.

Приложение

Здесь приведены варианты индивидуальных лабораторных работ, которые можно выполнять на ЭВМ параллельно с изучением рассматриваемого курса. Во всех вариантах заданий цепочки языка, для которого необходимо построить анализатор, заданы в виде правил, близких к расширенной форме Бэкуса-Наура. Если это не оговорено дополнительно, то используются следующие группы метасимволов: < ... > - нетерминал;
::= - разделитель левой и правой частей правил и обозначает: “это есть” или “состоит из”;
[...] - факультативный (необязательный) элемент;
{ ... } - итерация, т.е. элемент повторяется 0 или более раз;
? ... ¦ ... ¦ ... ? - альтернатива;

Используются также следующие сокращения: @ - произвольный идентификатор; k - константа, если не оговорено, то целая. Терминальные символы, а к ним здесь относятся и идентификаторы, и константы, выделены жирно.

Во всех заданиях необходимо для заданных цепочек построить автоматную грамматику, граф состояний, разработать алгоритм и реализовать программу синтаксического анализа на одном из языков высокого уровня. Предусмотреть максимальное число сообщений о синтаксических ошибках. Подготовить тесты проверяющие все ветви работы программы, рассматривающие все предельные случаи и т.д. Во всех заданиях выдавать предупреждающие сообщения при переполнении констант и в случаях, когда количество символов в идентификаторах больше 8-ми. Сравнение идентификаторов проводить по первым 8-ми символам.

Вариант 1

Построить синтаксический анализатор для цепочек автоматного языка операторов присваивания (ПЛ/1), имеющих вид:
<оператор присваивания> ::= {<метка>:}<левая часть>=<правая часть>;
<метка> ::= @
<левая часть> ::= @[(<список индексов>)]
<список индексов> ::= <индекс>{,<индекс>}
<индекс> ::= ? <левая часть>{<оп1><левая часть>} ¦ k ?
<оп1> ::= ? + ¦ - ¦ * ¦ / ?
<правая часть> ::= <операнд>{<оп2><операнд>}
<операнд> ::= ? <левая часть> ¦ k1 ?
где k1 - целая или действительная константа
<оп2> ::= ? <оп1> ¦ OR ¦ AND ?

Примеры правильных цепочек:
abc: ac123: a1(i+j/10,j*k,10,a2(1,i,15)-a2(1,2*i,7)+15,l)=
1234+a2(1,i,15)*1234.56E-3/aqs(3,a2(j,a2(1,2,3),15));
abcde=123; aaa=aaa OR bbb OR ccc AND 1;

Семантика: Сформировать безповторные упорядоченные списки идентификаторов, целых и действительных констант в числовой форме.

Сообщать об ошибках, если имена меток будут дублироваться или совпадать с именами переменных, а также в том случае, если массивы с одинаковыми именами будут иметь разную размерность или использоваться как имена скалярных переменных.

Собственно, рассматриваемые цепочки не являются автоматным языком, так как допускают вложенные скобки. Но при построении А-грамматики можно допустить произвольное количество открывающих и закрывающих скобок. Анализ правильности чередования скобок в этом случае следует возложить на семантические программы и сообщать об ошибках в случае нарушений в чередовании.

Вариант 2

Построить синтаксический анализатор для цепочек автоматного языка операторов описания типов (Модула-2), имеющих вид:

<описания типов> ::= TYPE <описание типа>; {<описание типа>;}

<описание типа> ::= @=? <простой тип> ¦ <массив> ¦ <множество> ¦ <запись> ¦ <указатель> ?

<простой тип> ::= ? CARDINAL ¦ INTEGER ¦ REAL ¦ CHAR ¦ SHORTCARD ¦ SHORTINT ¦ LONGCARD ¦ LONGINT ¦ LONGREAL ¦ BOOLEAN ¦ ADDRESS ¦ BYTE ¦ WORD ¦ <перечисление> ¦ <диапазон> ?

<перечисление> ::= (@{,@})

<диапазон> ::= [k1..k2]

<массив> ::= ARRAY <диапазоны> OF <тип>

<диапазоны> ::= <диап1>{,<диап1>}

<диап1> ::= ? <диапазон> ¦ @T1 ?

<тип> ::= ? <простой тип> ¦ @T ? ,

где @T - имя типа, определенного выше в анализируемом Вами операторе, @T1 - имя типа-дипазона

<множество> ::= SET OF <простой тип>

<запись> ::= RECORD <элемент> {;<элемент>} END

<элемент> ::= @{,@}: <тип>

<указатель> ::= POINTER TO <тип_1>

<тип_1> ::= ? <простой тип> ¦ @T2 ? ,

где @T2 - имя типа, определенного выше или ниже в анализируемом Вами операторе. Пример правильной цепочки:

```
TYPE  Color = ( Red, Blue, White, Black );
 diap = [10..25];  BitSet = SET OF WORD;
 Mas = ARRAY [0..100], [0..3], diap OF BitSet;
 PTab = POINTER TO Tab;
 Tab = RECORD a1, a2, a3: CARDINAL; col: Color;
       St: ARRAY [0..79] OF CHAR;
       left, right: PTab
      END;
 MTab = ARRAY [0..100] OF Tab;
```

Семантика: Сформировать список определяемых типов с указанием объема памяти, отводимого под тип.

Вариант 3

Построить синтаксический анализатор для цепочек автоматного языка операторов описания констант (Модула-2), имеющих вид:
<описания констант> ::= CONST <описание>; {<описание>;}
<описание> ::= @=<выражение>
<выражение> ::= ? <операнд>{<операция><операнд>} ¦ '<текст>'?
<операнд> ::= ? k ¦ @C ? ,
где k - целая или вещественная; @C - имя константы, определенной выше в анализируемом Вами операторе;
<текст> - произвольный набор символов.
<операция> ::= ? + ¦ - ¦ * ¦ / ¦ DIV ¦ MOD ?

Пример правильной цепочки:
CONST Abc = 1024 DIV 7 + 35 MOD 17; text = 'All right';
 Cde = 1234 - 32*13; rur = 3.14;
 rir= 123.*rur/12.3E-5+rur; Fg = Abc - Cde + 15;

Семантика: Сформировать список - таблицу констант с указанием имени, типа и значения. Сообщать об ошибках при переполнении констант, несовместимости типов и использовании неверных операций для конкретного типа.

Вариант 4

Построить синтаксический анализатор для цепочек автоматного языка операторов описания переменных (ПЛ/1), имеющих вид:
<описания переменных> ::= ? DCL ¦ DECLARE ? <список описаний>;
<список описаний> ::= <описание>{,<описание>}
<описание> ::= ? <переменная> [<атрибут>] ¦ (<список переменных>)<атрибут> ?
<переменная> ::= @[(<список размерностей>)]
<список размерностей> ::= <размерность>{,<размерность>}
<размерность> ::= k1[:k2]   ,
где k1 и k2 - целые числа со знаком
<атрибут> ::= ? BIN[ARY] FIXED ¦ FLOAT ¦ CHAR[ACTER](k) ?

Примеры правильных цепочек:
DCL   IND1, JIG, LOO, SUM, E123 BIN FIXED, III FLOAT, ST CHAR(10),
      STR CHARACTER(25), SUSPEED BINARY FIXED, IMAS(100),
      (ABC(10,-15:20,0:23), MICRO, MACRO(5:40), ERCOD) BIN FIXED;
DECLARE SSS(10:20,50) CHAR(1);

Семантика: По умолчанию, переменные, имена которых начинаются с букв I,J,K,L,M,N имеют тип BIN FIXED, остальные - FLOAT. Сообщать об ошибках, если k1 > k2, размер строки больше 256 символов и размер массива больше 64 Кбайт. Сформировать упорядоченный список-табл-

ицу переменных с указанием имени, типа, размерности, диапазона изменения индексов и объема памяти, выделяемой под переменную (кроме первых двух полей формируемой таблицы, все остальные поля в числовой форме). Напоминаем, что переменная типа BIN FIXED занимает 2 байта памяти, FLOAT - 4 байта, CHAR(1) - 1 байт.

Вариант 5

Построить синтаксический анализатор для цепочек автоматного языка операторов описания форматов (ПЛ/1), имеющих вид:

<описание форматов> ::= FORMAT(<элемент>{,<элемент>});

<элемент> ::= ? <формат> ¦ k1(<формат>{,<формат>}) ?

<формат> ::= ? A(k2) ¦ X(k2) ¦ SKIP(k1) ¦ F(k4[,k5]) ¦ E(k6,k7) ¦ <элемент> ?

где k1 - k7 - целые числа без знака.

Семантика: Сообщать об ошибках, если k1 > 10; k2 > 50; k4 > 33 и k4 < 4, если присутствует k5 ; k5 > k4-3; 8 > k6 > 37; k7 > k6-7. Осуществить вывод на экран информации по заданному формату: SKIP - переход на новую строку, с обозначением "|" для пустой строки, X - обозначить символом "-", A - "A", знак числа и порядка - "З", цифр мантиссы и порядка - "Ц".

Пример правильной цепочки:

FORMAT (X(5),3(A(2),X(2),F(3),X(1)), SKIP(3), X(7), 2(F(10,5)), X(2), E(10,4), SKIP(2), 4(X(10), A(5), F(5), 2(X(3), A(2), 3(F(2), A(1))), SKIP(1)), X(20),A(10));

Результат работы программы для данного оператора:

```
-----AA--ЗЦЦ-AA--ЗЦЦ-AA--ЗЦЦ-
|
|
-------ЗЦЦЦ.ЦЦЦЦЦЦЗЦЦЦ.ЦЦЦЦЦЦ--ЗЦ.ЦЦЦЦЕЗЦЦ
|
----------AAAAAЗЦЦЦЦ---AAЗЦAЗЦAЗЦA---AAЗЦAЗЦAЗЦA
----------AAAAAЗЦЦЦЦ---AAЗЦAЗЦAЗЦA---AAЗЦAЗЦAЗЦA
----------AAAAAЗЦЦЦЦ---AAЗЦAЗЦAЗЦA---AAЗЦAЗЦAЗЦA
----------AAAAAЗЦЦЦЦ---AAЗЦAЗЦAЗЦA---AAЗЦAЗЦAЗЦA
-------------------AAAAAAAAAA
```

Собственно, рассматриваемые цепочки не являются автоматным языком, так как допускают вложенные скобки. Но при построении А-грамматики можно допустить произвольное количество открывающих и закрывающих скобок. Анализ правильности чередования скобок в этом случае следует возложить на семантические программы и сообщать об ошибках в случае нарушений в чередовании.

Вариант 6

Построить синтаксический анализатор для цепочек автоматного языка операторов описания переменных (Модула-2), имеющих вид:

<описание переменных> ::= VAR <описание>;{<описание>;}

<описание> ::= @{,@}:[ARRAY <диапазон>{,<диапазон>} OF]<тип>

<диапазон> ::= [k1..k2]

<тип> ::= ? CARDINAL ¦ INTEGER ¦ REAL ¦ CHAR ¦ SHORTCARD ¦ SHORTINT ¦ LONGCARD ¦ LONGINT ¦ LONGREAL ¦ BOOLEAN ¦ ADDRESS ¦ BYTE ¦ WORD ?

Пример правильного оператора:

```
VAR  abc, A12C3,Ijk: CARDINAL;
       s,st,S: ARRAY [0..79] OF CHAR; abcdef: LONGINT;
       MasF: ARRAY [10..20],[5..19] OF BOOLEAN;     Re: REAL;
```

Семантика: Сообщать об ошибках, если k1 > k2, объем памяти под переменную больше 64 Кбайт. Сформировать упорядоченный по именам список-таблицу переменных с указанием имени, типа, размерности, диапазона изменения индексов и объема памяти, выделяемой под переменную (кроме первых двух символьных полей таблицы, все остальные поля представлять в числовой форме).

Вариант 7

Построить синтаксический анализатор для цепочек автоматного языка операторов заголовка цикла (Модула-2), имеющих вид:

<заголовок цикла> ::= FOR <параметр>:=<значение> TO <значение>
[BY <значение>] DO
<параметр> ::= @[[<список индексов>]]
<список индексов> ::= <индекс>{,<индекс>}
<индекс> ::= <операнд1>{<операция><операнд1>}
<операнд1> ::= ? @ ¦ k ?
<операция> ::= ? + ¦ - ¦ * ¦ MOD ¦ DIV ?
<значение> ::= <операнд2>{<операция><операнд2>}
<операнд2> ::= ? <параметр> ¦ k ?

Примеры правильных цепочек:

```
FOR  par[1,yy+23 MOD 7 DIV 2-1*3,kkk]:=ijk+aa[1,h-2]  TO  kkk*24 DIV 3
       BY aa[3,4]-3  DO
FOR  ijk:=1444-7 DIV 12 * 3  TO  12345  BY  23-1*5 DIV 2 DO
```

Семантика: Сформировать списки @ и k в числовой форме. Если все значения в операторе представляют собой выражения над константами, то определить сколько раз будет выполняться цикл.

Вариант 8

Построить синтаксический анализатор для цепочек автоматного языка операторов цикла (Модула-2), имеющих вид:

<цикл> ::= REPEAT {<присваивание>;} UNTIL <условие>;
<присваивание> ::= <левая часть>:=<правая часть>
<левая часть> ::= @[[<список индексов>]]
<список индексов> ::= <индекс>{,<индекс>}
<индекс> ::= <операнд1>{<операция1><операнд1>}
<операнд1> ::= ? @ ¦ k ?
<операция1> ::= ? + ¦ - ¦ * ¦ MOD ¦ DIV ¦ >> ¦ << ?
<правая часть> ::= <операнд2>{<операция1><операнд2>}
<операнд2> ::= ? <левая часть> ¦ k ?
<условие> ::= <правая часть1>{<операция2><правая часть1)}

<операция2> ::= ? = ¦ # ¦ < ¦ > ¦ >= ¦ <= ¦ <> ¦& ¦ AND ¦ OR ?
<правая часть1> ::= [? NOT ¦ ~ ?]<правая часть2>
<правая часть2>::=? (<правая часть>) ¦ (k IN <левая часть>) ?
Примеры правильных цепочек:
REPEAT UNTIL (7 IN abcd) & ~(5 IN aa[1,i+k MOD 4]);
REPEAT
aaa:=a+b-c[11,i*j DIV 2 -1,k+zzz123z]*1234 MOD 25;
bb[j+i>>2,12,34,ikj]:=bb[j+i>>2,12,34,ikj]+1;
i:=i-2; j:=i*k+3;
UNTIL (aaa) > (j+2) OR (i+c[1,i DIV 2,k+zzz123z] MOD 5)=(0) AND NOT(3 IN xx);

Семантика: Сформировать безповторные упорядоченные списки @ и k в числовой форме.

Вариант 9

Построить синтаксический анализатор для цепочек автоматного языка операторов цикла (Модула-2), имеющих вид:
<цикл> ::= WHILE <условие> DO {<присваивание>;} END;
<присваивание> ::= <левая часть>:=<правая часть>
<левая часть> ::= @[[<список индексов>]]
<список индексов> ::= <индекс>{,<индекс>}
<индекс> ::= <операнд1>{<операция1><операнд1>}
<операнд1> ::= ? @ ¦ k ?
<операция1> ::= ? + ¦ - ¦ * ¦ MOD ¦ DIV ¦ >> ¦ << ?
<правая часть> ::= <операнд2>{<операция1><операнд2>}
<операнд2> ::= ? <левая часть> ¦ k ?
<условие> ::= <правая часть1>{<операция2><правая часть1)}
<операция2> ::= ? = ¦ # ¦ < ¦ > ¦ >= ¦ <= ¦ & ¦ AND ¦ OR ?
<правая часть1> ::= [? NOT ¦ ~ ?]<правая часть2>
<правая часть2>::=? (<правая часть>) ¦ (k IN <левая часть>) ?
Примеры правильных цепочек:
WHILE (7 IN abcd) & ~(5 IN aa[1,i+k MOD 4]) DO END;
WHILE (aaa) > (j+2) OR (i+c[1,i DIV 2,k+zzz123z] MOD 5)=(0)
AND NOT(3 IN xx) DO
aaa:=a+b-c[11,i*j DIV 2 -1,k+zzz123z]*1234 MOD 25;
bb[j+i>>2,12,34,ikj]:=bb[j+i>>2,12,34,ikj]+1;
i:=i-2; j:=i*k+3;
END;

Семантика: Сформировать безповторные упорядоченные списки @ и k в числовой форме.

Вариант 10

Построить синтаксический анализатор для цепочек автоматного языка операторов ветвления (Модула-2), имеющих вид:
<ветвление> ::= IF <условие> THEN {<присваивание>;}
{ELSIF <условие> THEN {<присваивание>;}}
[ELSE {<присваивание>;}]

```
END;
<присваивание> ::= <левая часть>:=<правая часть>
<левая часть> ::= @[[<список индексов>]]
<список индексов> ::= <индекс>{,<индекс>}
<индекс> ::= <операнд1>{<операция1><операнд1>}
<операнд1> ::= ? @ ¦ k ?
<операция1> ::= ? + ¦ - ¦ * ¦ MOD ¦ DIV ¦ >> ¦ << ?
<правая часть> ::= <операнд2>{<операция1><операнд2>}
<операнд2> ::= ? <левая часть> ¦ k ?
<условие> ::= <правая часть1>{<операция2><правая часть1)}
<операция2> ::= ? = ¦ # ¦ < ¦ > ¦ >= ¦ <= ¦ & ¦ AND ¦ OR ?
<правая часть1> ::= [ ? NOT  ¦ ~ ? ]<правая часть2>
<правая часть2>::=? (<правая часть>) ¦ (k  IN <левая часть>) ?
Пример правильной цепочки:
IF (7 IN abcd) & ~(5 IN aa[1,i+k MOD 4])  THEN    l:=l+1;
ELSIF (aaa) > (j+2) OR (i+c[1,i DIV 2,k+zzz123z] MOD 5)=(0)
        AND NOT(3 IN xx)  THEN
   aaa:=a+b-c[11,i*j DIV 2 -1,k+zzz123z]*1234 MOD 25;
   bb[j+i>>2,12,34,ikj]:=bb[j+i>>2,12,34,ikj]+1;
   i:=i-2; j:=i*k+3;
ELSIF aabbcc  THEN
   i:=i+1;
ELSE     i:=j+b<<4*kkk[1,hg]; END;
```

Семантика: Сформировать безповторные упорядоченные списки @ и k в числовой форме.

Список рекомендуемой литературы

1. Ахо, А. Теория синтаксического анализа, перевода и компиляции. Том 1. Синтаксический анализ/ А. Ахо, Д. Ульман. - М.: Мир, 1978.
2. Ахо, А. Теория синтаксического анализа, перевода и компиляции. Том 2. Компиляция/ А.Ахо, Д. Ульман. - М.:Мир, 1978.
3. Братчиков, И.Л. Синтаксис языков программирования/И.Л. Братчиков. - М.: Наука, 1975.
4. Гилл, А. Введение в теорию конечных автоматов/А.Гилл. - М.: Наука, 1966.
5. Гинзбург, С. Математическая теория контекстно-свободных языков/С.Гинзбург. - М.: Мир, 1970.
6. Гладкий, А.В. Формальные грамматики и языки/А.В.Гладкий. - М.: Наука, 1973.
7. Грис, Д. Конструирование компиляторов для цифровых вычислительных машин/Д.Грис. - М.: Мир, 1975.
8. Гросс, М. Теория формальных грамматик/ М.Гросс, А. Лантен. - М.: Мир, 1971.
9. Донован, Д. Системное программирование/Д.Донован.- М.:Мир, 1975.
10.Лебедев, В.Н. Введение в системы программирования/В.Н.Лебедев.- М.:Статистика, 1975.

11. Льюис, Ф. Теоретические основы проектирования компиляторов/Ф.Льюис, Д.Розенкранц, Р.Стирнз - М.: Мир, 1979.
12. Семантика языков программирования. Сборник статей.- М.:Мир,1980.
13.Шамашов, М.А.Теория формальных языков. Грамматики и автоматы. Учебное пособие./М.А.Шамашов.-Самара: Самарский муниципальный комплекс непрерывного образования «Университет Наяновой», 1996, 92 с.
14.Хантер, Р. Проектирование и конструирование компиляторов/Р.Хантер. – М.:Финансы и статистика, 1984.
15.Хомский, Н. Формальные свойства грамматик/Н.Хомский// Кибернетический сборник, новая серия - вып. 2. - М.: ИЛ, 1966.
16.Хомский, Н. Алгебраическая теория контекстно-свободных языков/ Н. Хомский, М. Шютценберже // Кибернетический сборник, новая серия, вып. 3. - М.: ИЛ, 1966.
17.Хопгуд, Ф. Методы компиляции/Ф. Хопгуд.- М.:Мир, 1972.
18. Форстер, Дж. Автоматический синтаксический анализ/Дж. Фостер.-М.:Мир, 1972.
19.Шамашов, М.А. Основные структуры данных и алгоритмы компиляции: учеб. пособие/М.А.Шамашов.- Самара: Научно-внедренческая фирма «Сенсоры, модули, системы», 1999. –115 с.
20.Шамашов, М.А. Синтаксический анализ автоматных языков. Метод. указания/ М.А. Шамашов, Л.Ф. Штернберг. - Куйбышев: КуАИ, 1990.
21. Штернберг, Л.Ф. Теория формальных грамматик/Л.Ф. Штернберг. - Куйбышев: КуАИ, 1979.
22. Языки и автоматы: сб. статей. - М.: Мир, 1975.

Учебное издание

Чигарина Елена Ивановна
Шамашов Михаил Анатольевич

ТЕОРИЯ КОНЕЧНЫХ АВТОМАТОВ И ФОРМАЛЬНЫХ ЯЗЫКОВ

Учебное пособие

Научный редактор А. Г. Х р а м о в
Редакторская обработка В. С. Т е л е п о в а
Корректорская обработка В. С. Т е л е п о в а
Доверстка В. С. Т е л е п о в а

Подписано в печать 26.09.07. Формат 60х84 1/16.
Бумага офсетная. Печать офсетная.
Печ. л. 6,0.
Тираж 120 экз. Заказ . ИП -76/2007.

Самарский государственный
аэрокосмический университет.
443086 Самара, Московское шоссе, 34.

Издательство Самарского государственного
аэрокосмического университета.
443086 Самара, Московское шоссе, 34.