МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П.КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

**Е.В.Мясников**

# ОСНОВЫ ТРАНСЛЯЦИИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ
## Сборник задач для практических занятий

*Электронное учебное пособие*

Самара
2011

Автор: МЯСНИКОВ Евгений Валерьевич

Пособие представляет собой сборник задач для практических занятий. В сборник включены краткие теоретические сведения, необходимые для решения задач, задания для практических и самостоятельных занятий, подробно разобраны примеры решения типовых задач. Задачи способствуют закреплению теоретического материала и выработке практических навыков по дисциплине. В сборнике представлены задачи по всем темам курса, освещаемым на практических занятиях по дисциплине.

Пособие предназначено для студентов факультета информатики, направление 010400 – Прикладная математика и информатика, бакалавриат (010400.62)/магистратура (010400.68, магистерская программа – Технологии параллельного программирования и суперкомпьютинг).

# Оглавление

# 1. Алфавит, цепочка, язык

## *1.1 Краткие теоретические сведения*

*Алфавит* – конечное непустое множество символов, используемых в языке. Алфавит, как правило, обозначается заглавной латинской буквой A. Приведем несколько примеров.

Алфавит целых чисел восьмеричной системе счисления: A={0,1,2,3,4,5,6,7}.

Алфавит констант с фиксированной запятой: A={0,1,2,3,4,5,6,7,8,9, .}.

Алфавит языка C++: A={a,b,…z,A,B,…,Z,0,1,…,9,+, -, *, /, %, <, >, =, |, &, \, ', '', @, #, $, ^, ?, _, : , ; , . , (, ), [, ], {, }, !}.

Кроме алфавита в рамках курса будут, конечно же, использоваться и другие множества.

*Цепочка* – любая конечная последовательность, составленная из символов некоторого множества.

Так, 45098, 34.676, abc][#$99 –примеры цепочек, составленных из символов приведенных выше алфавитов.

Цепочки, как правило, обозначают буквами греческого алфавита, например:

α=12345, β=23.54.901, γ=abc[100]

Над цепочками символов определяют следующие операции:

1. *Определение длины* цепочки. Длина цепочки |α| равна количеству символов в цепочке α.

Например: |α|=5, |β|=9, |γ|=8

2. *Конкатенацию* цепочек. Конкатенацией цепочек α и β называется цепочка αβ, получаемая путем дописывания символов цепочки β вслед за символами цепочки α.

Например: Пусть α=34, β=56, тогда αβ=3456, βα=5634

3. *Итерацию* цепочки. Итерацией цепочки α называется цепочка $\alpha^i$, представляющая собой i раз повторенную цепочку α.

Например: Пусть α=34, β=abbcd, тогда $\alpha^3$=343434, $\beta^2$= abbcdabbcd,

В частности, если цепочка α состоит из i раз повторенного символа a, ее обозначают $a^i$. Например, $3^4$=3333.

Так, $(abc)^3$=abcabcabc, $a^3b^3b^3$=aaabbbccc, $a^2(bc)^2$=aabcbc, $a(b(cd)^2)^2$=abcdcdbcdcd

4. *Обращение* цепочки. Обращением цепочки α называется цепочка $\alpha^R$, полученная путем записи символов цепочки α в обратном порядке.

Например: Пусть α=12345, β=abbcd, тогда $\alpha^R$=54321, $\beta^R$=dcbba

Цепочку называют *пустой*, если она не содержит ни одного символа. Как правило, пустая цепочка обозначается ε. Для нее справедливо: |ε|=0, αε=εα=ε, $\varepsilon^i$=ε, $\alpha^0$=ε, $\varepsilon^R$=ε

В том случае, если U – некоторое множество символов, мы будем обозначать через $U^+$ множество всех возможных цепочек, составленных из символов множества U, за исключением пустой цепочки, а через U* - множество всех возможных цепочек, составленных из символов множества U, включая пустую цепочку: $U^*= U^+ \cup \{\varepsilon\}$.

С учетом сказанного, для любой цепочки α в некотором алфавите A справедливо $\alpha \in A^*$.

В том случае, если некоторая непустая цепочка γ составлена из символов множеств U и V, то $\gamma \in (U \cup V)^+$.

*Языком* в алфавите A называют некоторое множество цепочек, составленных из символов алфавита A.

В простых случаях язык A может быть задан путем перечисления цепочек, например

L={00,01,10,11} – язык двухразрядных чисел в двоичной системе счисления.

$L=\{\alpha, \beta, \alpha^R, \beta^R\}$ – язык, составленный из цепочек α и β и их обращений.

К сожалению, задание языков таким способом в большинстве случаев нецелесообразно, если вообще возможно, так как подавляющее большинство практически значимых языков либо содержат бесконечное число цепочек, либо перечисление цепочек таких языков представляется практически невыполнимой задачей.

## *1.2 Задания*

1. Опишите алфавит следующих языков:

A) Машинных кодов

B) Целочисленных констант

C) Констант с плавающей запятой

D) Идентификаторов

E) Арифметических выражений

2. Опишите путем перечисления цепочек следующие языки:

A) Всех двухсимвольных цепочек в алфавите {a,b}.

B) Всех трехсимвольных цепочек в алфавите {a,b,c}, не содержащих повторяющихся символов.

C) Всех цепочек в алфавите {a, b}, инвариантных относительно симметричного переворота общей длиной не более 4 символов

D) Язык, составленный из цепочек α,β,γ, всех возможных конкатенаций этих цепочек и пустой цепочки.

3. Опишите с использованием обозначений:

A) Множество всех непустых цепочек в алфавите {a,b}

B) Множество всех цепочек, составленных из символов множеств U,V,W, включая пустую цепочку

C) Множество всех непустых цепочек, составленных из символов некоторых множества U, V, и не содержащие символов из множества W.

D) Множество цепочек, составленных из трех одинаковых символов множества U.

E) Множество непустых цепочек, составленных из одинаковых символов множества U.

# 2. Грамматики

## *2.1 Краткие теоретические сведения*

*Грамматикой* называется четверка

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

где S – начальный символ грамматики,

$V_N$ - множество нетерминальных символов грамматики,

$V_T$ – множество терминальных символов грамматики, определяемое алфавитом языка,

R – непустое множество правил грамматики.

Каждое *правило* грамматики в общем случае записывается в виде $\alpha\rightarrow\beta$, где $\alpha$, $\beta$ – произвольные цепочки, составленные из терминальных и нетерминальных символов грамматики. Правила грамматики часто называют также правилами подстановки или продукциями. Цепочка $\alpha$ называется левой частью правила, $\beta$ – его правой частью.

Рассмотрим, например, грамматику со следующим набором правил:

$G_{2.1}$: $S\rightarrow AB$ $A\rightarrow AN$

$A\rightarrow\varepsilon$ $N\rightarrow 0$

$N\rightarrow 1$ $B\rightarrow .A$

$B\rightarrow\varepsilon$

Для этой грамматики S – *начальный символ* грамматики, $V_N=\{S,A,N,B\}$, $V_T=\{0,1,.\}$. Часто, говоря о грамматике, множества терминальных и нетерминальных символов не указывают, полагая, что они очевидны из набора правил грамматики.

В данном курсе для обозначения нетерминальных символов мы будем пользоваться прописными латинскими буквами. В тех случаях, когда это будет нецелесообразно или невозможно (например, если такие буквы входят в алфавит языка) для обозначения нетерминальных символов мы будем использовать угловые скобки.

Процесс *вывода* цепочки языка по заданной грамматике всегда начинается с начального символа грамматики $S\in V_N$. В процессе вывода происходит пошаговая замена частей уже выведенной цепочки (или цепочки целиком) в соответствии с правилами грамматики. При этом заменяемая часть цепочки должна совпадать с левой частью применяемого правила грамматики. Замена производится на правую часть применяемого правила.

Для приведенной выше грамматики $G_{2.1}$ мы можем выполнить следующий вывод:

**S**⇒A**B**⇒**A**.A⇒**A**N.A⇒**A**NN.A⇒**A**NNN.A⇒NNN.**A**⇒NNN.**A**N⇒NNN.**A**NN⇒

**N**NN.NN⇒1**N**N.NN⇒10**N**.NN⇒101.**N**N⇒101.0**N**⇒101.01

В процессе вывода на каждом шаге вывода мы заменяли один из нетерминальных символов цепочки (заменяемые символы выделены жирным) в соответствии с некоторым правилом грамматики. При этом мы использовали знак «⇒» для того, чтобы показать, что цепочка, стоящая слева от такого знака порождает цепочку, стоящую справа от знака. Более строгое определение выглядит следующим образом.

Цепочка $\alpha$ *непосредственно порождает* цепочку $\beta$ ($\alpha \Rightarrow \beta$), если цепочки $\alpha$ и $\beta$ представимы в виде: $\alpha = \varphi\mu\psi$, $\beta = \varphi\eta\psi$, и в грамматике имеется правило $\mu \to \eta$.

Довольно часто в подробном пошаговом описании вывода нет необходимости, поэтому несущественные или очевидные детали вывода можно опустить, при необходимости указывая количество шагов: $\alpha \Rightarrow^n \beta$. Это означает, что цепочка $\alpha$ порождает цепочку $\beta$ за n шагов. Так, для приведенного выше примера: $NNN.NN \Rightarrow^5 101.01$.

В том случае, если количество шагов не принципиально можно использовать обозначения «$\Rightarrow^+$» и «$\Rightarrow^*$». Более строго, цепочка $\alpha$ порождает цепочку $\beta$ ($\alpha \Rightarrow^+ \beta$), если существует последовательность непосредственных выводов $\alpha=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow ... \Rightarrow \gamma_n=\beta$, где $n>0$. Эта последовательность называется выводом длины n. В случае, если $\alpha$ порождает $\beta$ ($\alpha \Rightarrow^+ \beta$) или $\alpha$ равна $\beta$ ($\alpha = \beta$) записывают: $\alpha \Rightarrow^* \beta$.

В том случае, если некоторая цепочка $\alpha$ выводима из начального символа грамматики, такая цепочка называется *сентенциальной формой*: $S \Rightarrow^* \alpha$.

**Классификация грамматик Хомского.**

Грамматики разделяют на четыре класса.

1. *Автоматной* называется грамматика, все правила которой имеют вид:

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

где A, B – нетерминальные символы ($A,B \in V_N$), a – терминальный символ. ($a \in V_T$). Правила вида $A \to a$ называют заключительными (терминальными) правилами.

Так, автоматная грамматика для описания целочисленных констант имеет вид:

$S \to 0S|1S|\ldots|9S|0|1|\ldots|9$

Автоматная грамматика для описания идентификаторов:

$G_{2.2}$: $S \to aA|bA|\ldots|zA$

$A \to 0A|1A|\ldots|9A|0|1|\ldots|9|aA|bA|\ldots|zA|a|b|\ldots|z$

2. *Контекстно-свободной* называется грамматика, все правила которой имеют вид

$A \to \alpha$,

где A – нетерминальный символ грамматики ($A \in V_N$), $\alpha$ – произвольная цепочка ($\alpha \in (V_N \cup V_T)^*$).

Контекстно-свободная грамматика для описания идентификаторов может быть описана следующим образом:

$G_{2.3}$: S→LT

L→a|b|…|z

T→LT|DT|ε

D→0|1|…|9

3. *Контекстно-зависимой* называется грамматика, все правила которой имеют вид

φAψ→φαψ,

где A – нетерминальный символ грамматики ($A \in V_N$), α –непустая цепочка ($\alpha \in (V_N \cup V_T)^+$), а цепочки φ и ψ, называемые контекстом, - произвольные цепочки ($\varphi, \psi \in (V_N \cup V_T)^*$).

4. *Рекурсивно-перечислимыми* называют грамматики, на вид правил которых не накладывается никаких ограничений:

α→β,

где α, β - произвольные цепочки α, ($\beta \in (V_N \cup V_T)^*$).

Нас будут интересовать лишь первые два рассмотренных класса грамматик, так как последние два класса грамматик в курсе не рассматриваются.

Как можно видеть, в правилах автоматных и контекстно-свободных грамматиках грамматик в левой части всегда стоит нетерминальный символ, а правые части могут существенно отличаться. В автоматных грамматиках правая часть правила всегда начинается с терминального символа, за которым может следовать только нетерминальный символ. В контекстно-свободных грамматиках на правую часть не накладывается никаких ограничений.

При выводе в автоматных грамматиках цепочки содержат не более одного нетерминального символа, который всегда стоит в конце сентенциальной формы. В том случае, когда единственный нетерминальный символ заменяется на терминальный символ в соответствии с терминальным символом вывод завершается. Например, для $G_{2.2}$: S⇒gA⇒g2A⇒g23.

В контекстно-свободных грамматиках заменяться любой из нетерминальных символов сентенциальной формы. Так, в следующем выводе по $G_{2.3}$:

**S**⇒**L**T⇒g**T**⇒g**D**T⇒g2**T**⇒g2**D**T⇒g24**T**⇒g24

заменяется всегда самый левый нетерминальный символ и такой вывод называется *левым* (*левосторонним*) выводом.

В следующем случае заменяется самый правый нетерминал и такой вывод называется *правым* (*правосторонним*):

**S**⇒L**T**⇒LD**T**⇒LDD**T**⇒LD**D**⇒L**D**4⇒**L**24⇒g24

Вообще говоря, в каком порядке заменялись нетерминальные символы сентенциальной формы, не имеет большого значения. Гораздо важнее уметь определять какую часть сентенциальной формы породил тот или иной нетерминальный символ. И для этой цели форма записи в виде последовательности непосредственных выводов – не самое удобное средство. По этой причине при выводе и анализе цепочек, выводимых по контекстно-свободным грамматикам, распространение получили *деревья вывода* (*синтаксические деревья*).

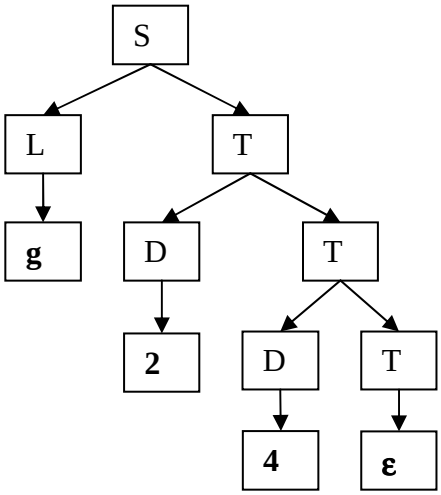



Рисунок 2.1 – Дерево вывода для грамматики $G_{2.3}$

На рисунке представлено дерево вывода соответствующее представленным выше последовательностям непосредственных выводов для грамматики $G_{2.3}$. Отметим, что дерево вывода не определяет точный порядок применения правил, но более наглядно отображает изменения, происходившие в структуре выводимой цепочки.

Понять, какая же цепочка была выведена по дереву вывода можно, воспользовавшись левосторонним алгоритмом обхода дерева. При этом символы терминальной цепочки будут содержаться в терминальных узлах (листьях) дерева. Они помечены на рисунке жирным шрифтом.

Дерево вывода можно построить и для автоматной грамматики. В этом случае дерево при выводе будет разрастаться в одну сторону и иметь вид, подобный показанному на рис. 2.2.

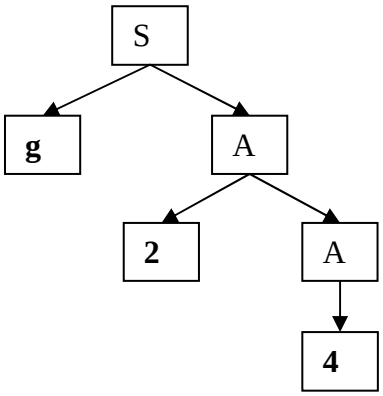



Рисунок 2.2 – Дерево вывода для автоматной грамматики $G_{2.2}$

В приведенном выше примере два различных вывода по грамматике порождали одну и ту же терминальную цепочку, и выводам соответствовало одно и то же дерево вывода. Однако несложно привести пример, когда двум различным выводам, порождающим одну и ту же терминальную цепочку, будут соответствовать различные деревья вывода.

$G_{2.4}$: $S \rightarrow SS|i$

В этой грамматике можно выполнить следующие выводы:

$S \Rightarrow \mathbf{S}S \Rightarrow i\mathbf{S} \Rightarrow i\mathbf{S}S \Rightarrow ii\mathbf{S} \Rightarrow iii$

S⇒**S**S⇒**S**SS⇒i**S**S⇒ii**S**⇒iii

Этим выводам будут соответствовать различные деревья вывода, показанные на рис. 2.3. Этот факт свидетельствует о том, что одна и та же цепочка, порожденная исходной грамматикой, может иметь различную структуру. А так как именно со структурой цепочки связывается смысл цепочки, то такие цепочки называются *двусмысленными* или *неоднозначными*.

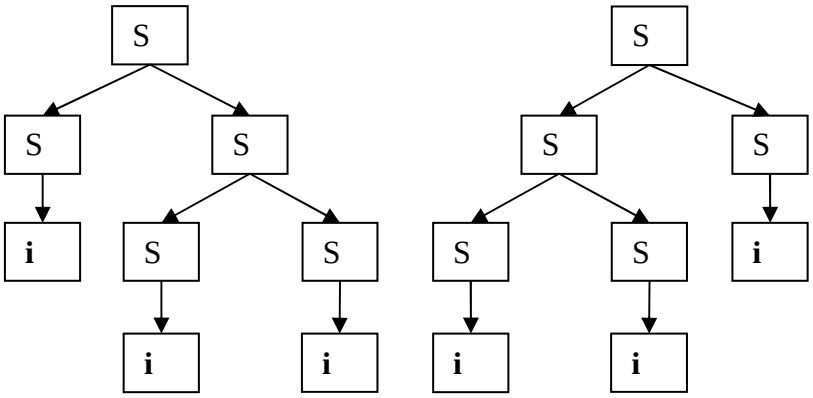



Рис. 2.3 – Деревья вывода для неоднозначной грамматики $G_{2.4}$

Итак, цепочка, порождаемая грамматикой, называется *двусмысленной* (*неоднозначной*), если для нее существует более одного дерева вывода.

Грамматика, порождающая неоднозначные цепочки, называется *неоднозначной* (*двусмысленной*). В том случае, если двусмысленных цепочек порождено быть не может, грамматика называется однозначной.

На первый взгляд, неоднозначность грамматики $G_{2.4}$ не порождает никаких проблем. Чтобы понять, чем плохи неоднозначные грамматики, немного изменим $G_{2.4}$:

$G_{2.5}$: S→S+S|S*S|i

Если под i здесь понимать произвольный идентификатор, то приведенная выше грамматика описывает арифметические выражения над переменными с операциями сложения и умножения. По $G_{2.5}$ можно выполнить следующие выводы:

S⇒**S+**S⇒i+**S**⇒i+**S***S⇒i+i***S**⇒i+i*i

S⇒**S***S⇒**S+**S*S⇒i+**S***S⇒i+i***S**⇒i+i*i

Синтаксические деревья, соответствующие приведенным выше выводам представлены на рис.2.4.

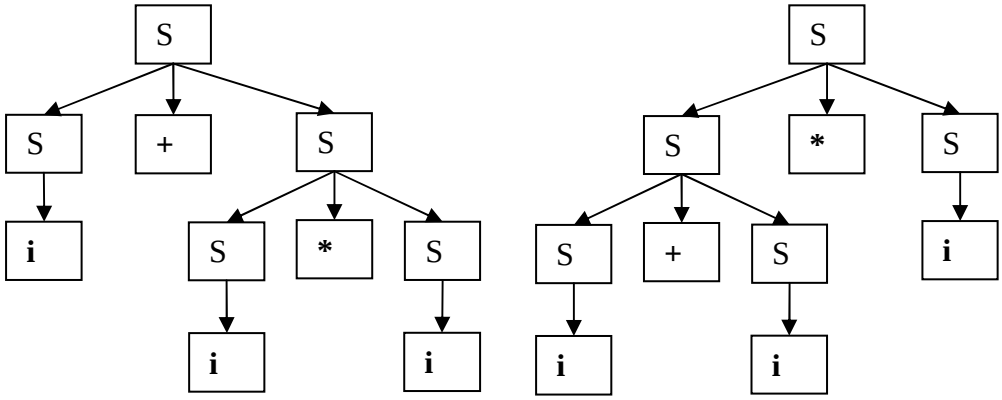



Рис. 2.4 – Деревья вывода для неоднозначной грамматики $G_{2.5}$

Теперь, в том случае, если мы захотим произвести вычисления по синтаксическим деревьям, показанным на рис. 2.4 мы обнаружим, что в первом случае вычисления будут выполняться правильно (сначала умножение, затем сложение), а во втором случае – неправильно, так как привычный приоритет операций будет нарушен (сначала сложение, затем умножение). Таким образом, взяв грамматику $G_{2.5}$ за основу при построении анализатора арифметических выражений, мы столкнемся с определенными проблемами.

Грамматику $G_{2.5}$ легко изменить, чтобы избавиться от неоднозначности. С использованием такой грамматики составить двусмысленные цепочки невозможно:

$S \rightarrow S+T|T$

$T \rightarrow T*i|i$

К сожалению, определить по виду правил, однозначна ли контекстно-свободная грамматика, удается не всегда. Более того, в общем случае невозможно определить, является ли произвольная контекстно-свободная грамматика однозначной.

Однако, для того, чтобы показать неоднозначность грамматики, достаточно привести пример двусмысленной цепочки, порождаемой этой грамматикой. Но даже в том случае, когда точно известно, что некоторая грамматика неоднозначна, нет гарантий того, что удастся найти эквивалентную ей однозначную грамматику. Ситуацию усугубляет и тот факт, что в общем случае невозможно определить, существует ли вообще однозначная грамматика.

В приведенном выше примере мы довольно просто избавились от неоднозначности в грамматике $G_{2.5}$, причиной которой являлась двусторонняя рекурсия. Вообще говоря, двусторонняя рекурсия довольно часто является причиной неоднозначности. В этом случае замена двусторонней рекурсии на одностороннюю помогает избавиться от неоднозначности.

## *2.2 Примеры*

**Задание.** Построить автоматную грамматику, порождающую цепочки вида $(xy)^n z^k$, $n \geq 0, k > 0$.

**Решение.** Вид правил автоматных грамматик определяет и способ построения таких грамматик для описания языков. Построение правил выполняется пошагово. Так как в правой части правил автоматных грамматик первым стоит терминальный символ, то для формирования правил на каждом шаге:

1) опираясь на описание языка, анализируем цепочки, которые могут встретиться на данном шаге;

2) выбираем те терминальные символы, которые могут встретиться в начале рассматриваемых цепочек;

3) обозначаем соответствующие остатки цепочек нетерминальными символами и формируем для них правила рассматриваемым способом. В том случае, если цепочка на данном шаге может оборваться, используем терминальные правила.

В рассматриваемом примере на первом шаге нам могут встретиться цепочки вида:

xy…z… или z…. Поэтому первой парой правил будет

$S \rightarrow xA|zB$

Замечаем, что сразу за z цепочка может оборваться, поэтому добавляем правило

$S \rightarrow z$

Описываемые нетерминалом A цепочки имеют вид yxy…z… или yz… То есть цепочки начинаются с y, а к их остаткам предъявляются те же требования, что и к исходным цепочкам. Поэтому в правой части правила для A содержится нетерминал S:

$A \rightarrow yS$

Описываемые нетерминалом B цепочки имеют вид z…, где к остатку, следующему за z, предъявляются те же требования, что и к цепочкам, обозначенным нетерминалом B. Поэтому

$B \rightarrow zB$

Замечая, что сразу за z цепочка может оборваться, добавляем терминальное правило

$B \rightarrow z$

Таким образом, искомая грамматика выглядит следующим образом:

$S \rightarrow xA|zB|z$

$A \rightarrow yS$

$B \rightarrow zB|z$

**Задание.** Построить контекстно-свободную грамматику, порождающую цепочки вида:

$(xy)^n z^k$, $n \geq 0, k > 0$.

**Решение.** При построении контекстно-свободной грамматики:

1) опираясь на описание языка, анализируем цепочки, которые могут встретиться на данном шаге;

2) разбиваем цепочки на части, обозначая нетерминальным символом каждую из частей;

3) формируем правила для каждой из выделенных частей рассматриваемым способом.

Из описания языка очевидно, что его цепочки состоят из двух независимых частей: $(xy)^n$ и $z^k$. Поэтому первое правило грамматики имеет следующий вид:

$S \rightarrow AB$

Далее, цепочки, порождаемые нетерминалом A, представляют собой последовательность xy, за которой может следовать произвольное количество таких же последовательностей. Поэтому правило для A будет рекурсивным:

$A \rightarrow xyA$

Замечая, что при n=0 нетерминал A должен порождать пустую строку, добавляем

$A \rightarrow \varepsilon$

Для нетерминала B также применяем рекурсивное правило:

$B \rightarrow zB$

Учитывая, что хотя бы один символ z должен быть выведен из B, добавляем

$B \rightarrow \varepsilon$

Таким образом, получаем грамматику:

$S \rightarrow AB$

$A \rightarrow xyA|\varepsilon$

$B \rightarrow zB|\varepsilon$

**Задание.** Построить КС-грамматику для цепочек, составленных из букв русского алфавита и инвариантных относительно симметричного поворота. Например, анна, ого, абба, казак и т.д.

**Решение**. По сути дела, цепочки языка могут быть описаны следующим образом: $\alpha\beta\alpha^R$, где $\alpha \in (а,б,\ldots,я)^*$, $\beta \in \{\varepsilon,а,б,\ldots,я\}$

Несмотря на то, что в цепочке явно выделяются три части ($\alpha$, $\beta$ и $\alpha^R$), разбивать ее на три независимые компоненты было бы неверным, так $\alpha$ и $\alpha^R$ тесно связаны. Вместо этого заметим, что в цепочках указанного вида символы, стоящие на симметричных относительно центра цепочки позициях, обязаны совпадать. В частности, должны совпадать первый и последний символы цепочек, то есть: а…а, б…б, в…в и т.д. Если при этом учесть, что к средней части цепочки предъявляются такие же требования, получим рекурсивное правило

$S \rightarrow aSa|бSб|\ldots|яSя$

Далее, допустимы как односимвольные, так и двухсимвольные цепочки. Поэтому

S→а|б|…|я

S→аа|бб|…|яя

Таким образом, искомая грамматика

S→аАа|бАб|…|яАя|а|б|…|я|аа|бб|…|яя

Заметим, если в решении группу правил S→аа|бб|…|яя заменить на правило S→ε, получим грамматику, допускающую пустые цепочки, о чем в задании сказано не было.

## *2.3 Задания*

1. Постройте автоматную грамматику, порождающую цепочки следующего вида:

A) {xy,yx}

B) {xyz,xzy,yxz,yzx,zxy,zyx}

C) $x^n$, n>0

D) $(xy)^n$, n>0

E) $(xyz)^n$, n>0

F) $x^n y^m$, n,m>0

G) $x^n y^m z^k$, n,m,k>0

H) {00,01,10,11}

I) {000,001,010,011,100,101,110,111}

J) все цепочки в алфавите {0,1}, не содержащие двух идущих подряд нулей (или единиц)

K) все цепочки в алфавите {0,1}, содержащие четное число нулей (или единиц)

L) $\alpha_1\beta_1\alpha_2\beta_2\ldots\alpha_n\beta_n$, $\alpha_i \in \{x,y\}, \beta_i \in \{0,1\}^*, i=1\ldots n$

2. Постройте контекстно-свободную грамматику, порождающую цепочки следующего вида. Покажите вывод цепочки длиной не менее 6 символов (там, где это возможно) по построенной грамматике (постройте цепочку вывода или дерево вывода)

A) {xyz,xzy,yxz,yzx,zxy,zyx}

B) $(xyz)^n$, n>0

C) $x^n y^n$, n>0

D) $x^n y^n z^m$, n,m,>0

E) $x^n y^m z^m$, n,m>0

F) $x^n y^m z^n$, n,m>0

G) $x^n y^m z^k$, n,m,k>0

H) $x^n y^m x^m y^n$, n,m>0

I) $x^n y^n x^m y^m$, n,m>0

J) $\alpha\beta\alpha^R$, $\alpha \in \{x,y\}^+$

K) все цепочки в алфавите {x,y}, содержащие равное количество x и y

L) все цепочки в алфавите {x,y}, содержащие неравное количество символов x и символов y

M) все цепочки в алфавите {x,y}, содержащие строго больше символов x, чем символов y

N) все цепочки в алфавите {x,y}, содержащие в два раза больше символов x, чем y

O) $\alpha_1\beta_1\alpha_2\beta_2\ldots\alpha_n\beta_n$, $\alpha_i \in \{x,y\}, \beta_i \in \{0,1\}^*, i=1\ldots n$

P) $x^n y^{2n}$

Q) $01x^n y^{2n}101$

Q) $x^{2n+1} y^{2n}$

3. Опишите словами языки, порождаемые грамматиками, содержащими следующие правила:

A) S→S,i|i

B) S→1A

A→0A|0

C) S→<S>|<>

D) S→NSN|ε

N→0|1|…|9

4. Составьте грамматику, описывающую 32 разрядные целые числа в 16-ой системе счисления.

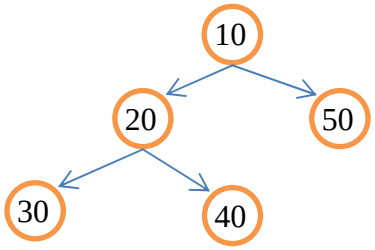
5. Составьте грамматику, порождающую следующие языки:

A) Язык, состоящий из выражений, составленных из операндов i и операций + и –.

B) Язык, состоящий из всех выражений в полной скобочной записи с операндами i,c и операциями +,*.

6. Известно, что древовидные структуры данных могут быть описаны с использованием скобочной формы записи. В одной из таких форм нетерминальные узлы записываются в угловых скобках, при этом сразу за открывающей скобкой записывается хранящаяся в узле информация, а за ней после двоеточия перечисляются дочерние узлы с использованием запятой в качестве разделителя. Для терминальных узлов указывается только связанная с ними информация без скобок. Составьте грамматику, порождающую соответствующий язык для следующих случаев:

A) Во всех узлах дерева хранятся целочисленные константы, например: <10:<20:30,40>,50>.



B) Во нетерминальных узлах дерева хранятся идентификаторы, а в терминальных - целочисленные константы. Например: <node10:<node20:30,40>,50>.

C) С каждым нетерминальным узлом может быть связан набор целочисленных атрибутов, записываемых после двоеточия в виде <идентификатор>=<значение> и перечисляемых через пробел.

Например: <node1: a=1 b=10<node2 a=2 b=20:30,40>,50>.

7. Составьте следующую грамматику, порождающую все возможные последовательности парных открывающих и закрывающих квадратных скобок:

A) Контекстно-свободную грамматику без ограничения на глубину вложенности скобок.

B) Автоматную грамматику с глубиной вложенности скобок не более, чем 1.

C) Автоматную грамматику с глубиной вложенности скобок не более, чем 2.

8. Скажите, является ли приведенная ниже грамматика однозначной (недвусмысленной), почему?

A) S→AA
A→aa|a

B) S→A+S|A*S|A
A→a

C) S→(S)|A|SBS
A→id
B→+|-|*|/

D) S→id|(A)
A→A+S|S

E) S→id|(S)|(S+S)|(S*S)

F) S→-S|S-i|i

G) S→not S|(S)|S and S|S or S

3. Для неоднозначных грамматик из задания 2 постройте неоднозначные цепочки и деревья вывода. Предложите пути модификации таких грамматик таким образом, чтобы они стали однозначными. Постройте деревья вывода найденных ранее неоднозначных цепочек для модифицированных грамматик.

# 3. Синтаксический анализ автоматных языков

## *3.1 Краткие теоретические сведения*

Альтернативой формой задания языка по отношению к автоматной грамматике является описание языка с использованием конечного автомата. *Конечный автомат* представляет собой пятерку

$(\Sigma, Q, q_0, \delta, F)$,

где $\Sigma$ - конечное множество допустимых входных символов (алфавит),

$Q$ – множество состояний автомата,

$q_0$ – множество начальных состояний автомата,

$\delta$ - функция переходов,

$F$ - множество допускающих состояний автомата.

По автоматной грамматике конечный автомат может быть построен элементарно. Для этого можно руководствоваться следующими правилами:

1) производят замену терминальных правил $A \rightarrow a$ на правила вида $A \rightarrow aF$, где $F$ – новый нетерминальный символ;

2) множеству терминальных символов грамматики $V_T$ ставят в соответствие алфавит конечного автомата $\Sigma$, множеству нетерминальных символов грамматики $V_N$ - множество состояний автомата $Q$, начальному символу грамматики $S$ - начальное состояние автомата $q_0$, нетерминальному символу $F$ - допускающее состояние автомата $F$;

3) Функцию перехода $\delta$ определяют следующим образом: $\delta(A,a) := \{B|A \rightarrow aB\}$.

Для грамматики описания идентификаторов

$S \rightarrow aA|bA|\ldots|zA$

$A \rightarrow 0A|1A|\ldots|9A|0|1|\ldots|9|aA|bA|\ldots|zA|a|b|\ldots|z$

функция перехода, представленная в виде таблицы переходов, имеет вид (начальное состояние помечено стрелкой сверху, допускающее состояние – единицей снизу):

| | ↓ S | A | F |
|---|---|---|---|
| a,b…z | A | A, F | |
| 0,1…9 | | A, F | |

1

В контексте построения программы анализа большое значение имеет то, в какой форме представлена автоматная грамматика. Дело в том, что программа может быть

непосредственно написана лишь по грамматике, представленной во вполне детерминированной форме, в то время как исходная грамматика часто представлена в недетерминированной форме. Дадим определения формам грамматики.

Говорят, что автоматная грамматика представлена в *детерминированной* форме, если для любого заданного нетерминального символа A и любого заданного терминального символа a существует не более одного правила $A \to aX$ (X – нетерминальный символ). В противном случае автоматная грамматика представлена в *недетерминированной* форме.

Говорят, что автоматная грамматика представлена во *вполне детерминированной* форме, если для любого заданного нетерминального символа A и любого заданного терминального символа a существует единственное правило $A \to aX$ (X – нетерминальный символ).

Аналогичные определения можно дать и для конечных автоматов. Конечный автомат является *детерминированным* (ДКА), если множество начальных $q_0$ состояний состоит из единственного начального состояния, а функция переходов $\delta(q,a)$ возвращает не более одного состояния для любых q, a. В противном случае мы имеем дело с *недетерминированным* конечным автоматом (НКА).

Тот факт, что исходная автоматная грамматика представлена в недетерминированной форме, некритичен, так как для такой грамматики можно найти эквивалентную автоматную грамматику в детерминированной форме. Часто такое преобразование грамматик можно выполнить, исходя из общих соображений, но существует формализованный алгоритм такого преобразования.

Учитывая, что для каждой автоматной грамматики существует эквивалентный ей конечный автомат, сформулируем этот алгоритм, как алгоритм перехода от НКА к ДКА.

**Алгоритм перехода от НКА к ДКА**

Пусть $(\Sigma, Q, q_0, \delta_N, F_N)$ – исходный НКА, $(\Sigma, P, p_0, \delta_D, F_D)$ – искомый ДКА, эквивалентный исходному. Тогда алгоритм перехода может быть записан следующим образом:

1. Создадим заготовку таблицы переходов искомого автомата и добавим в нее столбец, соответствующий начальному состоянию $p_0$ ДКА, помечая его в шапке таблицы множеством начальных состояний НКА: $p_0 := \{q_i | q_i \in q_0\}$.

2. В том случае, если для некоторого состояния ДКА $p_x$, помеченного множеством состояний НКА $Q_x$ функция переходов еще не вычислена, вычисляем ее следующим образом:

$\delta_D(p_x,a) := \{\delta_N(q_i,a) | q_i \in Q_x\}$, $a \in \Sigma$.

Другими словами для каждого символа *a* алфавита Σ, помечаем соответствующую ячейку $\delta_D(p_x,a)$ ДКА множеством всех состояний $\delta_N(q_i,a)$ НКА, достижимых по символу a из тех состояний $q_i$ НКА, которыми помечен текущий рассматриваемый столбец $p_x$ ДКА.

3. Для каждого вычисленного на шаге 2 множества $\delta_D(p_x,a)$ проверяем, существует ли в таблице ДА столбец, обозначенный этим множеством. Если такого столбца нет, то создаем его, включая в множество состояний ДКА новое состояние, помеченное как $\delta_D(p_x,a)$.

4. Повторяем шаги 2 и 3 до тех пор, пока все переходы $\delta_D(p_x,a)$ не будут вычислены и порождение новых столбцов (состояний) ДКА не будет прекращено.

5. Определяем заключительные состояния ДКА. Для этого помечаем столбец $p_x$ ДКА как допускающее состояние, если он содержит хотя бы одно допускающее состояние НКА, т.е.

$F_D=\{p_i|q_j\in p_i\cup q_j\in F_N\}$.

Переход от детерминированной формы ко вполне детерминированной форме осуществляется достаточно просто. Для этого в грамматику вводят так называемый ошибочный нетерминальный символ E и для каждого нетерминального символа A ($A\neq E \cup A\neq F$) добавляют правила $A\rightarrow aE$ для тех терминальных символов a, которые еще не участвуют в правилах для A:

$R=R\cup\{ A\rightarrow aE \mid (A\neq E) \cup (A\neq F) \cup (A\rightarrow aB \notin R) \}$, $A,B\in V_N$, $a\in V_N$.

В таблице переходов ДКА это аналогично введению дополнительного ошибочного состояния и заполнению пустых клеток переходами в новое ошибочное состояние.

Для написания программы удобно использовать граф переходов, построенный по грамматике или ДКА.

## *3.2 Пример*

**Задание**. Построить ДКА, эквивалентный следующему НКА:

↓ ↓

| | A | B | C | D |
|---|---|---|---|---|
| a | A,B | | A | |
| b | | B,D | | |
| c | | | C,B | |

1

**Решение**. Построение ДКА, эквивалентного исходному начнем с определения нового начального состояния, помечая его множеством начальных состояний НКА. Так как

начальными состояниями НКА являются состояния A и C, то начальное состояние НКА пометим, как {A,C}.

↓

| | {A,C} |
|---|---|
| a | |
| b | |
| c | |

Вычислим функцию переходов ДКА для состояния {A,C}. Так как из состояния A по символу a были достижимы состояния {A,B}, а из состояния C – состояние A, то функция переходов для {A,C} по символу a будет иметь значение {A,B}.

↓

| | {A,C} |
|---|---|
| a | {A,B} |
| b | |
| c | {C,B} |

Далее из состояний A и C по символу b переходы отсутствуют. По символу c НКА может перейти в C и B, поэтому функция переходов ДКА будет иметь значение {C,B}.

Далее проверяем, существуют ли столбцы, помеченные новыми состояниями ДКА: {A,B} и {A,C}.

Так как таких столбцов еще нет, вводим их:

↓

| | {A,C} | {A,B} | {C,B} |
|---|---|---|---|
| a | {A,B} | | |
| b | | | |
| c | {C,B} | | |

После расчета функции переходов для вновь введенных состояний ДКА получим:

↓

| | {A,C} | {A,B} | {C,B} |
|---|---|---|---|
| a | {A,B} | {A,B} | {A} |
| b | | {B,D} | {B,D} |
| c | {C,B} | | {C,B} |

Повторяя проделанные шаги для появившихся состояний, приходим к автомату:

↓

| | {A,C} | {A,B} | {C,B} | {A} | {B,D} |
|---|---|---|---|---|---|
| a | {A,B} | {A,B} | {A} | {A,B} | |
| b | | {B,D} | {B,D} | | {B,D} |
| c | {C,B} | | {C,B} | | |

Так как допускающим состоянием НКА было состояние D, то единственным допускающим состоянием ДКА будет состояние {B,D}. Таким образом, исходный НКА преобразован к следующему эквивалентному ДКА:

↓

| | {A,C} | {A,B} | {C,B} | {A} | {B,D} |
|---|---|---|---|---|---|
| a | {A,B} | {A,B} | {A} | {A,B} | |
| b | | {B,D} | {B,D} | | {B,D} |
| c | {C,B} | | {C,B} | | |

1

### *3.3 Задания*

1. Представьте автоматную грамматику в виде графа:

A) S→0A
A→xB
B→0C|1C|…|9C|aC|bC|…|fC
C→0|1|…|9|a|b|…|f

B) S→0S|1S|…|9S|0|1|…|9

C) S→aA|bA|…|zA
A→aA|bA|…|zA|[B
B→0C|1C|…|9C
C→0C|1C|…|9C|,B|]

D) S→aS|aA
A→bA|bB|b
B→cB|c

2. Скажите, какие грамматики из задания 1 представлены в недетерминированной форме. Предложите, как можно модифицировать грамматики для приведения их ко вполне детерминированной форме. Постройте таблицу переходов соответствующих недетерминированного и детерминированного автоматов.

3. Постройте конечный автомат, допускающий следующие цепочки в алфавите цифр десятичной системы счисления. Скажите, является ли построенный автомат детерминированным.

A) Все числа, записанные без лидирующих нулей.

B) Число 456, записанное, возможно, с лидирующими нулями.

C) Все числа, кроме числа 456.

D) Все числа, меньшие 456.

E) Все числа, большие или равные 456.

F) Числа 4, 45 и 456.

G) Все числа, содержащие хотя бы одну цифру 5.

H) Все числа, не содержащие ни одной цифры 5.

I) Все числа, содержащие ровно 2 цифры 5.

J) Все числа, содержащие более двух цифр 5.

K) Все числа, содержащие четное число цифр 5.

L) Все числа, содержащие нечетное число цифр 5 и четное число цифр 6.

M) Все числа, в которых каждая вторая цифра – цифра 5.

N) Все числа, в которых после каждой цифры 5 стоит цифра 6. Допускается, что цифры 5 может и вовсе не быть в числе.

O) Все числа, в которых перед каждой цифрой 5 стоит цифра 4. Допускается, что цифры 5 может и вовсе не быть в числе.

P) Все числа, в которых каждая цифра 5 окружена цифрами 4(перед цифрой 5) и 6 (после цифры 5). Допускается, что цифр 5 может и вовсе не быть в числе.

Q) Все числа, начинающиеся и заканчивающиеся цифрой 5.

R) Все числа, в которых между двумя цифрами 5 стоит ровно 3 любых других цифры.

S) Все числа, в которых количество цифр 5 кратно 3.

4. Опишите цепочки, допускаемые следующим недетерминированным конечным автоматом. Постройте эквивалентный детерминированный конечный автомат. Постройте графы переходов, соответствующие исходному и построенному автоматам.

A) ↓

| | A | B | C |
|---|---|---|---|
| 0 | A | B | C |
| 1 | A,B | B,C | C |

1

B) ↓

| | A | B | C | D |
|---|---|---|---|---|
| 0 | | | D | |
| 1 | B,C | A | | C |

1 1

C) ↓

| | A | B | C |
|---|---|---|---|
| 0 | B | | |
| 1 | | A,C | A |

1

D) ↓

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | | A | | C | |
| 1 | B,C | | D,E | | |

1

E) ↓

| | A | B | C | D |
|---|---|---|---|---|
| 0 | | B,C | | D |
| 1 | A,B | | C,D | |

1 1

F) ↓

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | | | A | E | |
| 1 | B,D | C | | | D |

1 1

G) ↓ ↓

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | | A,E | | D,E | |
| 1 | B | | C,D | | |

1

G) ↓ ↓

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | | C | | E | |
| 1 | B | | A,E | D | |

1

# 4. Синтаксический анализ контекстно-свободных языков

## *4.1 Краткие теоретические сведения*

Задачей синтаксического анализа является получение ответа на вопрос о том, принадлежит ли анализируемая цепочка заданному языку. Однако при трансляции уметь определять факт принадлежности, вообще говоря, недостаточно. Необходимо еще уметь определять структуру анализируемой цепочки, что дает возможность определять смысловое значение цепочки. Для контекстно-свободных языков структура цепочки адекватно отражается деревом вывода цепочки. В связи с этим, можно считать, что задача синтаксического анализа контекстно-свободных языков представляет собой задачу восстановления дерева вывода входной цепочки.

Восстановление дерева вывода может вестись *сверху-вниз*, то есть от корня дерева (начального символа грамматики) к листьям (терминальным символам цепочки). Такой подход получил название *нисходящего синтаксического анализа*. Однако возможен и другой подход – восстановление *снизу-вверх*, также называемое *восходящим синтаксическим анализом*. В существующих автоматических методах анализа нашли свое отражение оба подхода: как нисходящий, так и восходящий.

То, каким образом может выполняться нисходящий и восходящий синтаксический анализ, рассмотрим на примере. Пусть задана грамматика

S→A[B]

A→i

B→i,B|c,B|i|c

Левый вывод цепочки в этой грамматике может выглядеть следующим образом:

**S**⇒**A**[B]⇒i[**B**]⇒i[c,**B**]⇒i[c,i]

Правый вывод той же самой цепочки:

**S**⇒A[**B**]⇒A[c,**B**]⇒**A**[c,i]⇒i[c,i]

Синтаксическое дерево, соответствующее представленным выше выводам:

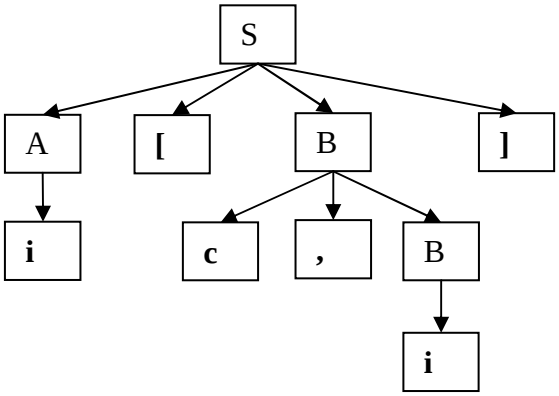
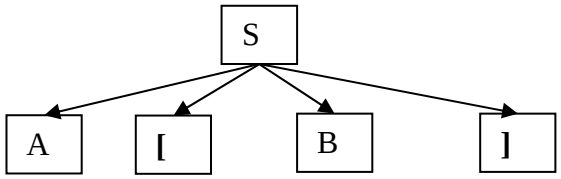



Рис. 4.1 – Дерево вывода цепочки

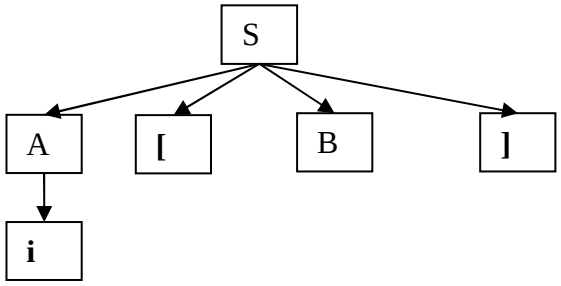
Попробуем теперь восстановить вывод цепочки нисходящим способом.




**S**⇒A[B]

Рис. 4.2 – Шаг 1 нисходящего анализа

На первом шаге анализа мы можем предположить, что было применено первое правило грамматики, так как других альтернатив нет. То есть мы можем построить часть дерева, изображенную слева.

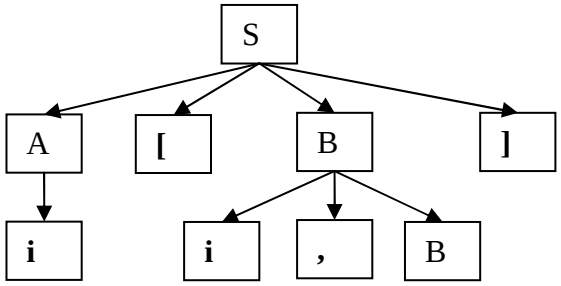



**S**⇒**A**[B]⇒i[B]

Рис. 4.3 – Шаг 2 нисходящего анализа

Теперь единственной альтернативой для A является i, то есть дерево принимает вид, показанный на рис. 4.3.

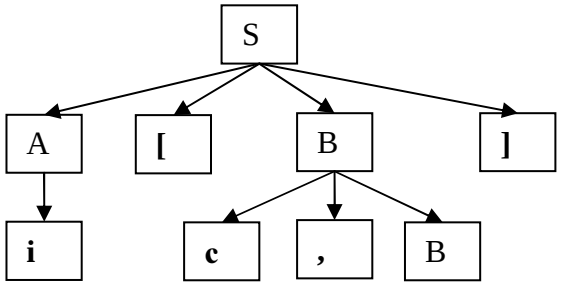
Здесь мы обнаруживаем, что первые два выведенных терминала в полученной сентенциальной форме «**i[**B]» совпадают с таковыми во входной строке «**i[**c,i]».




**S**⇒**A**[B]⇒i[**B**]⇒i[i,B]

Рис. 4.4 – Шаг 3 нисходящего анализа

Далее, для B мы имеем несколько альтернатив: i,B|c,B|i|c. Пробуя первую из них (рис.4.4), убеждаемся, что выведенная в этом случае часть сентенциальной формы «**i[i,**B]» отличается от входной строки «**i[c,**i]».

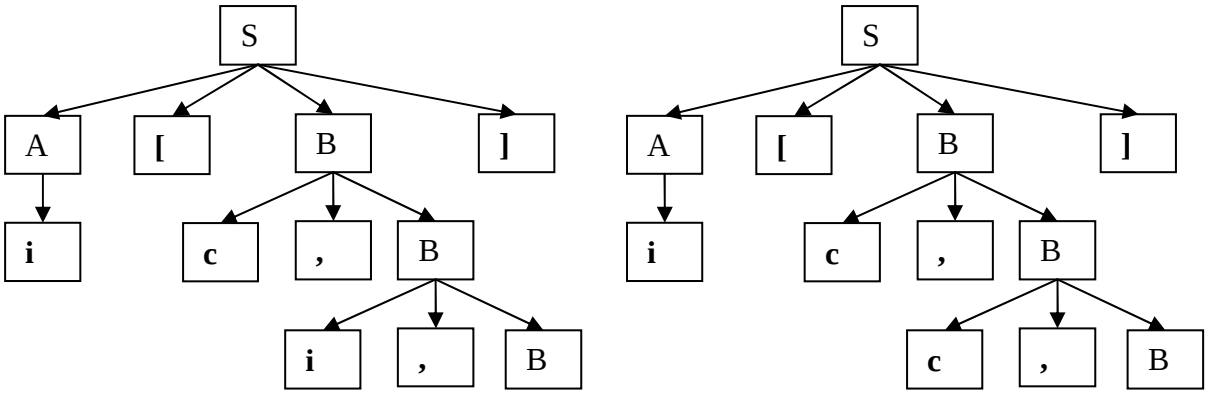



По этой причине пробуем вторую альтернативу (рис.4.5) и убеждаемся, что противоречий в сентенциальной форме «**i[c,**B]» с входной строкой «**i[c,**i]»не выявлено.

**S**⇒**A**[B]⇒i[**B**]⇒i[c,B]

Рис. 4.5 – Шаг 4 нисходящего анализа

Далее мы заменяем нетерминал B на возможные альтернативы i,B|c,B получая соответствующие деревья вывода (рис. 4.6), и убеждаемся в том, что полученные сентенциальные формы («**i[c,i,**B]» и «**i[c,c,**B]») не соответствуют входной цепочке «**i[c,i]**»:




**S**⇒**A**[B]⇒i[**B**]⇒i[c,**B**]⇒i[c,i,B]  **S**⇒**A**[B]⇒i[**B**]⇒i[c,**B**]⇒i[c,c,B]

Рис. 4.6 – Шаги 5 и 6 нисходящего анализа

Наконец, выбирая альтернативу B→i, приходим к терминальной цепочке «**i[c,i]**», совпадающей с входной цепочкой.

Таким образом, мы восстановили левый вывод цепочки **S**⇒**A**[B]⇒i[**B**]⇒i[c,**B**]⇒i[c,i]. При этом на каждом шаге мы сравнивали выведенные слева терминальные символы с символами входной строки и пытались заменить самый левый нетерминал сентенциальной формы. Это и обуславливает тот факт, что восстановлен левый вывод цепочки. С таким же успехом мы могли бы восстановить правый вывод, заменяя самые правые нетерминалы сентенциальной формы и сравнивая правые терминальные символы получаемых цепочек.

Восходящий анализ контекстно-свободных языков выполняется иным образом. Для этого просматриваем входную цепочку и пытаемся найти в ней основу для свертки - подцепочку, которую можно в соответствии с каким-либо правилом грамматики свернуть к нетерминалу грамматики.

Во входной цепочке i[c,i] таких основ может быть найдено сразу несколько. Более того, наличие в грамматике правил с одинаковыми правыми частями приводит к тому, что основы могут быть свернуты к разным нетерминалам.

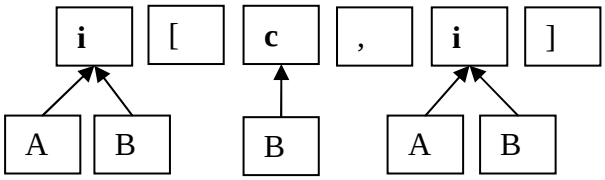
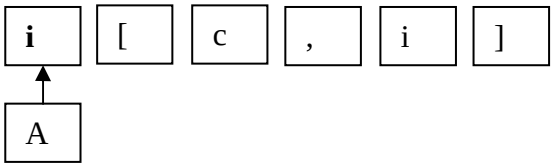



Рис. 4.7 – Возможные варианты свертки при восходящем анализе

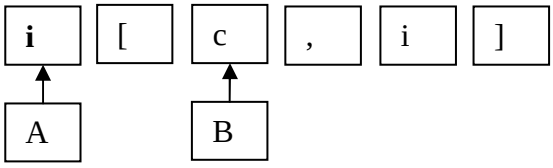
Тем не менее, установим порядок просмотра цепочки слева-направо и будем пытаться выполнять сверку последовательно по найденным правилам грамматики.




**A**[c,i]⇒i[c,i]

Рис. 4.8 – Шаг 1 восходящего анализа

Тогда на первом шаге находим терминал i, который можно свернуть по правилам A→i и B→i. Сворачивая по первому из правил, получим часть синтаксического дерева, представленную на рис. 4.6.

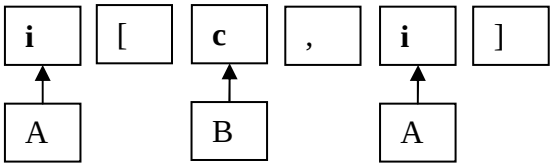



A[**B**,i]⇒**A**[c,i]⇒i[c,i]

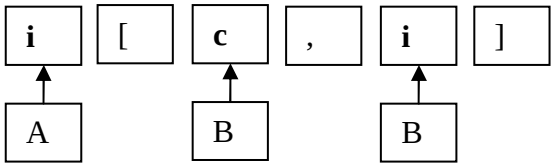
Рис. 4.9 – Шаг 2 восходящего анализа

Далее, обнаруживаем основу c, и сворачиваем ее по единственно возможному правилу (рис.4.9).

Далее обнаруживаем, что терминал i может быть свернут по правилам A→i и B→i и строим соответствующие деревья (рис. 4.10).



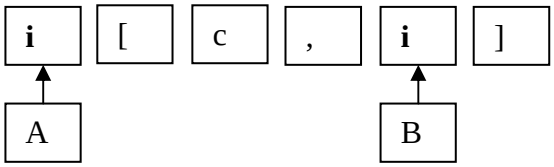

A[B,**A**]⇒A[**B**,i]⇒**A**[c,i]⇒i[c,i]




A[B,**B**]⇒A[**B**,i]⇒**A**[c,i]⇒i[c,i]

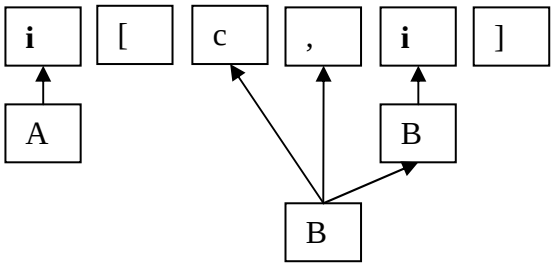
Рис. 4.10 – Шаги 3 и 4 восходящего анализа

На этом этапе, впрочем, обнаруживаем, что в полученных цепочках A[B,A] и A[B,B] основы выделены быть не могут.



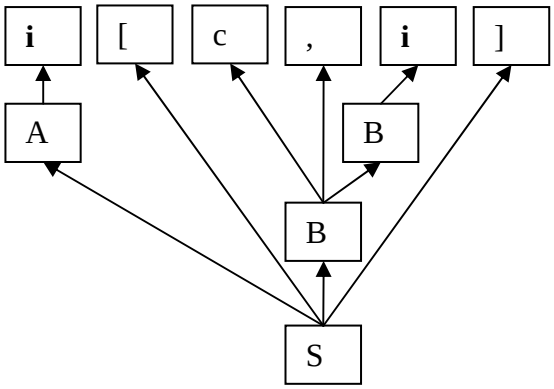

A[c,B]⇒A[c,i]⇒i[c,i]

Рис. 4.11 – Шаг 5 восходящего анализа




A[B]⇒A[c,B]⇒A[c,i]⇒i[c,i]

Рис. 4.11 – Шаг 6 восходящего анализа




S⇒A[B]⇒A[c,B]⇒A[c,i]⇒i[c,i]

Рис. 4.12 – Результат восходящего анализа

Предполагаем, что свертка на предыдущем шаге была неверной и приходим к фрагменту дерева, представленного на рис. 4.11.

Просматривая слева-направо цепочку A[**c,B**] обнаруживаем очередную основу для свертки по правилу B→c,B (рис. 4.11).

Единственно возможный вариант свертки здесь приводит нас к начальному символу грамматики

Итак, мы восстановили правый вывод исходной цепочки. При этом мы просматривали цепочку слева-направо, пытаясь отыскать подходящую основу для свертки. В том случае, если бы мы просматривали цепочку справа-налево, мы бы восстановили левый вывод исходной цепочки.

Из приведенного примера видны основные проблемы, встающие при нисходящем и восходящем анализе. В первом случае на каждом шаге нам нужно уметь правильно выбирать, какое из правил грамматики применить для замены самого левого нетерминала.

Во втором случае на каждом шаге нам нужно уметь правильно определять основу для свертки. В приведенных примерах, как для нисходящего, так и для восходящего анализа мы, по сути, перебирали по очереди все возможные варианты, откатываясь назад в случае неудачи. Такой подход к восходящему или нисходящему анализу называется *синтаксическим анализом с возвратами* и, конечно же, может быть реализован программно. Главный недостаток синтаксического анализа с возвратами – большое время работы, которое растет экспоненциально с ростом длины анализируемой цепочки.

## *4.2 Задания*

1. Выполните нисходящий синтаксический анализ по заданным грамматикам для приведенных ниже цепочек. При этом постарайтесь восстановить, как левый, так и правый вывод указанных цепочек.

A) Грамматика: S→(S)|S or S|S and S|0|1
   Цепочка: (0 or 1) and 1 or 0

B) Грамматика: S→[N]S|[N]
   N→NN|0|1|…|9
   Цепочка: [12][3][45]

C) Грамматика: S→iRS|i
   R→.|^|^.
   Цепочка: i.i^.i

D) Грамматика: S→t L;
   L→V,L|V
   V→i|i=c
   Цепочка: t i=c,i,i;

E) Грамматика: S→S+i|S*i|i
   Цепочка: i+i*i+i

F) Грамматика: S→SS+|SS*|i
   Цепочка: iii+*i+

2. Путем последовательного выделения основ для свертки в соответствие с грамматиками, представленными в задании 1, восстановите дерево вывода для приведенных в задании 1 цепочек. Постарайтесь восстановить, как левый, так и правый вывод указанных цепочек.

# 5. Нисходящий синтаксический анализ КС-языков. LL(1) грамматики

## *5.1 Краткие теоретические сведения*

*LL(k) грамматиками* называют контекстно-свободные грамматики, для которых допустимо применение детерминированного нисходящего синтаксического анализа, выполняющего восстановление левого канонического вывода цепочки, с просмотром цепочки слева направо и предварительным просмотром не более чем k символов относительно текущей позиции во входной строке.

Построение анализаторов LL(k) грамматик для k>1 достаточно сложно и трудоемко. По этой причине на практике получили распространение лишь LL(1) грамматики. Однако, даже с использованием LL(1) грамматик можно описать многие языки программирования. Отметим также, что такие классы грамматик, как детерминированные автоматные грамматики и грамматики рекурсивного спуска входят в класс LL(1) грамматик.

Прежде, чем дать формальное определение LL(1) грамматикам и перейти к рассмотрению метода их анализа, введем две важные функции, которые будем использовать в дальнейшем: $First_k(\ldots)$ и $Follow_k(\ldots)$.

Пусть α – некоторая цепочка. Тогда $First_k(\alpha)$ – множество укороченных до k символов цепочек, выводимых из α и состоящих только из терминальных символов:

$First_k(\alpha) = \{\beta \mid \alpha \Rightarrow^* \beta\gamma, |\beta|=k, \beta \in V_T^*, \gamma \in (V_T \cup V_T)^*\} \cup \{\beta \mid \alpha \Rightarrow^* \beta, |\beta|<k, \beta \in V_T^*\}, \alpha \in (V_T \cup V_T)^*$

Пусть A – некоторый нетерминальный символ. Тогда $Follow_k(A)$ – множество укороченных до k символов цепочек, которые состоят из терминальных символов и могут следовать непосредственно за A в сентенциальных формах:

$Follow_k(A) = \{\alpha \mid S \Rightarrow^* \varphi A \alpha \psi, |\alpha|=k, \alpha \in V_T^*, \varphi,\psi \in (V_T \cup V_T)^*\} \cup$

$\{\alpha \mid S \Rightarrow^* \varphi A \alpha, |\alpha|<k, \alpha \in V_T^*, \varphi \in (V_T \cup V_T)^*\}, \alpha \in (V_T \cup V_T)^*.$

В частности, при k=1:

$First_1(\alpha)$ – множество терминальных символов (односимвольных цепочек), которые могут стоять в начале цепочек, выводимых из α:

$First_1(\alpha) = \{b \mid \alpha \Rightarrow^* b\gamma, b \in V_T, \gamma \in (V_T \cup V_T)^*\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\}, \alpha \in (V_T \cup V_T)^*$

$Follow_1(A)$ – множество терминальных символов (односимвольных цепочек), которые могут следовать непосредственно за A в сентенциальных формах:

$Follow_1(A)=\{a \mid S \Rightarrow^* \varphi A a \psi, a \in V_T, \varphi,\psi \in (V_T \cup V_T)^*\} \cup \{\varepsilon \mid S \Rightarrow^* \varphi A, \varphi \in (V_T \cup V_T)^*\}, \alpha \in (V_T \cup V_T)^*.$

Для определения $First_1(\alpha)$ можно воспользоваться следующим алгоритмом:

1. Если α – пустая строка, то формируем множество, состоящее из пустой цепочки:

**Если** $\alpha=\varepsilon$, **то** $First_1(\alpha) := \{\varepsilon\}$.

2. Если α начинается с терминального символа a, то формируемое множество из цепочки a:

**Если** $\alpha=a\beta$, **то** $First_1(\alpha) := \{a\}$, $a \in V_T$, $\beta \in (V_T \cup V_T)^*$.

3. Если α начинается с некоторого нетерминального символа A, то формируем множество $First_1(\alpha)$, как результат применения $First_1()$ для всех правых частей правил, где в левой части стоит A:

**Если** $\alpha=A\gamma$, **то** $First_1(\alpha) := First_1(\beta_1) \cup First_1(\beta_2) \cup \ldots \cup First_1(\beta_N)$,

$A \in V_N$, $\gamma \in (V_T \cup V_T)^*$, $A \rightarrow \beta_1$, $A \rightarrow \beta_2$, …, $A \rightarrow \beta_N$.

В том случае, если $First_1(\alpha)$ включает пустую цепочку, а цепочка γ - непустая ($\gamma \neq \varepsilon$), то исключаем ε из $First_1(\alpha)$ и добавляем в формируемое множество $First_1(\gamma)$:

**Если** $\varepsilon \in First_1(\alpha)$, **то** $First_1(\alpha) = First_1(\alpha) \backslash \{\varepsilon\} \cup First_1(\gamma)$

Определить $Follow_1(A)$ можно следующим образом:

1. Просматриваем правила грамматики и для каждой продукции вида $X \rightarrow \varphi A \psi$ включаем во множество $Follow_1(A)$ множество $First_1(\psi) \backslash \{\varepsilon\}$:

**Если** $(X \rightarrow \varphi A \psi) \in R$, **то** $Follow_1(A) := Follow_1(A) \cup (First_1(\psi) \backslash \{\varepsilon\})$.

2. В том случае, если для какого либо правила $X \rightarrow \varphi A \psi$ пустая цепочка входит во множество $First_1(\psi)$, то включаем во множество $Follow_1(A)$ множество $Follow_1(X)$:

**Если** $(X \rightarrow \varphi A \psi) \in R$ **и** $\varepsilon \in First_1(\psi)$, **то** $Follow_1(A) := Follow_1(A) \cup Follow_1(X)$.

С использованием введенных функций дадим определение LL(1) грамматики.

Контекстно-свободная грамматика является *LL(1) грамматикой*, если для любого нетерминального символа A, для которого имеется множество правил $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_N$, выполняется следующее:

1) Множества $First_1$ для правых частей правил попарно не пересекаются: $First_1(\alpha_i) \cap First_1(\alpha_j) = \varnothing$, где $\alpha_j$, $\alpha_j \neq \varepsilon$;

2) Если среди правых частей правил есть приводимые к пустой цепочке, то множество $Follow_1(A)$ не должно пересекаться ни с одним из множества $First_1$ для правых частей правил: Если $\varepsilon \in First_1(\alpha_i)$, то $Follow_1(A) \cap First_1(\alpha_j) = \varnothing$.

Собственно распознаватель для LL(1) грамматики строится как автомат с магазинной памятью с одним состоянием. Управляющая таблица такого автомата показывает:

1) какая цепочка символов должна быть записана в магазин,

2) должен ли произойти переход к следующему символу входной цепочки,

в зависимости от:

а) символа, находящегося в вершине магазина,

б) текущего просматриваемого символа во входной цепочке.

Управляющая таблица строится следующим образом. Столбца таблицы помечаются символами, которые могут встретиться во входной цепочке (терминальными символами). Строки таблицы помечаются нетерминальными и терминальными символами. Ячейка (i,j) таблицы, стоящая на пересечении i-ой строки и j-го столбца заполняется следующим образом:

1. Если $A_M$ – нетерминальный символ, для которого в своде правил грамматики есть правило $A_M \to \alpha$, то в ячейку $(A_M;a_B)$ заносится правая часть этого правила (α), если терминальный символ $a_B$, входит во множество $First_1(\alpha)$. Ячейка помечается таким образом, что сдвиг головки чтения автомата не производится.

2. Если $A_M$ – нетерминальный символ, для которого в своде правил грамматики есть аннулирующее правило $A \to \varepsilon$, то в ячейку $(A_M;a_B)$ заносится ε, если терминальный символ $a_B$, входит во множество $Follow_1(A_M)$. Ячейка помечается таким образом, что сдвиг головки чтения автомата не производится.

2. Если $a_M$ – терминальный символ и $a_M=a_B$, то в ячейку $(a_M;a_B)$ заносится пустая строка. Ячейка помечается таким образом, что производится сдвиг головки чтения автомата на следующий символ.

3. Все незаполненные ячейки будут указывать на ошибки во входной строке при анализе.

| | | | **Терминальные символы грамматики** | | | |
|---|---|---|---|---|---|---|
| | | | **$a_1$** | **$a_2$** | **…** | **$a_m$** |
| **Вершина магазина** | **Нетерминальные символы** | **$A_1$** | α, если $(A_M \to \alpha) \in R$ и $a_B \in First_1(\alpha)$, ε, если $(A_M \to \varepsilon) \in R$ и $a_B \in Follow_1(A_M)$, без сдвига | | | |
| | | **$A_2$** | | | | |
| | | **…** | | | | |
| | | **$A_N$** | | | … | |
| | **Терминальные символы грамматики** | **$a_1$** | ε, сдвиг | | … | |
| | | **$a_2$** | | ε, сдвиг | … | |
| | | **…** | … | … | … | … |
| | | **$a_m$** | | | … | ε, сдвиг |

Рис. 5.1 – Управляющая таблица LL(1) распознавателя

Анализ входной строки с использованием построенного LL(1) распознавателя выполняется описанным далее образом. В начальной конфигурации в магазине содержится единственный символ – начальный символ грамматики, головка чтения обозревает первый

символ входной ленты. При анализе цепочки, записанной на входной ленте, автомат должен выполнить последовательность шагов, приводящую к заключительной конфигурации. В заключительной конфигурации требуется, чтобы память распознавателя была пуста, а головка сошла с правого края входной ленты. Собственно один шаг выполняется под управлением таблицы распознавателя следующим образом. Анализируется символ, обозреваемый головкой чтения $a_B$ и символ, находящийся в вершине магазина $a_M$. По информации в ячейке управляющей таблицы, находящейся на пересечении строки $a_M$ и столбца $a_B$, определяется, какие символы будут помещены в магазин вместо $a_M$ и следует ли сдвигать головку чтения автомата.

1. Если вершина магазина содержит терминальный символ, совпадающий с символом, обозреваемым головкой чтения на входной ленте ($a_B=a_M$), то символ $a_M$ выталкивается из магазина(вместо него в магазин ничего не помещается, что обозначается в таблице символом ε), а входная головка сдвигается на следующий символ (обозначается подписью «*сдвиг*» в соответствующей ячейке таблицы).
2. Если вершина магазина содержит некоторый нетерминальный символ $A_M$, то этот символ выталкивается из магазина и в магазин записывается строка символов, стоящая на пересечении строки $A_M$ и столбца $a_B$ (если соответствующая ячейка содержит ε, то в магазин ничего не помещается). Положение входной головки при этом не меняется.
3. Ситуация при которой соответствующая ячейка управляющей таблицы пуста, свидетельствует об ошибке во входной цепочке. Анализ в этом случае завершается.

Отметим, что возможны также ситуации, при которых головка чтения еще не дошла до правого края входной ленты, а магазин оказался пуст, или, наоборот, головка сошла с ленты, а в магазине остались некоторые символы. Такие ситуации также трактуются как ошибки во входной строке, свидетельствующие о лишних символах во входной строке, или о неожиданном конце входной строки, соответственно.

## *5.2 Пример*

**Задание**. Построить LL(1) распознаватель для следующей грамматики.

S→SLE|E

L→or|and

E→VRV|V

R→<|>

V→i|c

**Решение**. Очевидно, грамматика содержит леворекурсивное правило (S→SLE) и не является LL(1) грамматикой. Действительно, множества $First_1$ для первых правил грамматики пересекаются (совпадают):

S→SLE: $First_1(SLE)= First_1(E)= First_1(V)=\{i,c\}$

S→E: $First_1(E)=\{i,c\}$

Устраним левую рекурсию путем замены этих правил на следующие:

S→EA

A→LEA|ε

Хотя множества $First_1$ для правил нового нетерминала A не пересекаются

A→LEA: $First_1(LEA)= First_1(L)=\{or,and\}$

A→ε: $First_1(ε)=\{ε\}$

наличие в множестве $First_1$ пустой строки приводит к необходимости вычисления множества $Follow_1$ для нетерминала A. Далее, так как A является крайним правым символом сентенциальной формы (S⇒EA), то для вычисления $Follow_1(A)$ удобно ввести в грамматику терминальный символ конца цепочки (⊥) и расширить грамматику следующим правилом для нового начального символа грамматики (S’):

S’→S⊥

Отметим, что при решении реальных задач символ конца цепочки, как правило, присутствует естественным образом. Роль такого символа выполняют символы перевода строки, возврата коретки, разделители операторов и т.п. В любом случае, можно считать, что его роль выполняет признак конца входного потока, показывающий, что все символы строки прочитаны.

Далее отметим, что хотя правила для нетерминала E не содержат рекурсию, наличие одинаковых подцепочек в левых частях правил сразу же выводит грамматику из класса LL(1) грамматик, так как множества $First_1$ для правил нетерминала E будут совпадать.

Выполним левую факторизацию для нетерминала E.

E→VB

B→RV|ε

Таким образом, грамматика принимает следующий вид:

S’→S⊥

S→EA

A→LEA A→ε

L→or L→and

E→VB

B→RV B→ε

R→< R→>

V→i V→c

Вычислим множества $First_1$ для всех правил грамматики и убедимся, что для любого взятого нетерминала множества не пересекаются.

S'→S⊥: $First_1(S⊥)=First_1(S)=First_1(E)=First_1(V)=\{i,c\}$

S→EA: $First_1(EA)= First_1(E)=\{i,c\}$

A→LEA: $First_1(LEA)= First_1(L)=\{or,and\}$

A→ε: $First_1(ε)=\{ε\}$

L→or: $First_1(or)=\{or\}$

L→and: $First_1(and)=\{and\}$

E→VB: $First_1(VB)=First_1(V)=\{i,c\}$

B→RV: $First_1(RV)= First_1(R)=\{<,>\}$

B→ε: $First_1(ε)=\{ε\}$

R→<: $First_1(<)=\{<\}$

R→>: $First_1(>)=\{>\}$

V→i: $First_1(i)=\{i\}$

V→c: $First_1(c)=\{c\}$

Как было сказано выше, наличие во множестве $First_1$ пустой строки приводит к необходимости вычисления множества $Follow_1$ для соответствующего нетерминала. В нашем случае, такая ситуация возникает по причине наличия в грамматике аннулирующих правил для нетерминалов A и B.

$Follow_1(A)= Follow_1(S)=\{⊥\}$

$Follow_1(B)= Follow_1(E)=First(A)/\{ε\}\cup Follow(A)=\{or,and,ε\}/\{ε\}\cup\{⊥\}=\{or,and,⊥\}$

Полученные множества $Follow_1$ не пересекаются с множествами $First_1$, построенными для правил грамматики соответствующих нетерминалов (см. соответственно правила A→LEA, A→ε и B→RV, B→ε), поэтому грамматика относится к классу LL(1) грамматик.

Построим управляющую таблицу LL(1) анализатора. Столбцы таблицы помечаем терминальными символами грамматики, строки – нетерминальными и терминальными символами грамматики. Для каждого правила грамматики A→α заносим α во все ячейки находящиеся на пересечении строки, помеченной нетерминалом A, и столбцов, помеченных терминалами, входящими во множество $First_1(α)$ (ячейки записаны в таблице обычным шрифтом). В случае аннулирующего правила A→ε заносим ε во все ячейки, находящиеся на пересечении строки, помеченной нетерминалом A, и столбцов, помеченных терминалами,

входящими во множество $Follow_1(A)$ (помечены в таблице подчеркиванием). Наконец, в ячейки таблицы, находящие на пересечении строки и столбца одного и того же терминала записываем ε (помечены в таблице курсивом).

| | ⊥ | **or** | **and** | < | > | **i** | **c** |
|---|---|---|---|---|---|---|---|
| **S’** | | | | | | S⊥ | S⊥ |
| **S** | | | | | | EA | EA |
| **A** | ε | LEA | LEA | | | | |
| **L** | | or | and | | | | |
| **E** | | | | | | VB | VB |
| **B** | ε | ε | ε | RV | RV | | |
| **R** | | | | < | > | | |
| **V** | | | | | | i | c |
| **⊥** | *ε, сдвиг* | | | | | | |
| **or** | | *ε, сдвиг* | | | | | |
| **and** | | | *ε, сдвиг* | | | | |
| < | | | | *ε, сдвиг* | | | |
| > | | | | | *ε, сдвиг* | | |
| **i** | | | | | | *ε, сдвиг* | |
| **c** | | | | | | | *ε, сдвиг* |

Проверим с использованием построенного распознавателя следующую цепочку на принадлежность языку, порождаемому рассмотренной грамматикой

i or c<i and i>c

Для проверки будем пользоваться таблицей, приведенной ниже. В левом столбце будем записывать остаток входной строки, начиная с символа, обозреваемого головкой чтения. В правом – содержимое магазина (вершина магазина находится слева).

| **Входная строка** | **Магазин** |
|---|---|
| i or c<i and i>c | S’ |
| i or c<i and i>c | S⊥ |
| i or c<i and i>c | EA⊥ |
| i or c<i and i>c | VBA⊥ |
| i or c<i and i>c | iBA⊥ |
| or c<i and i>c | BA⊥ |
| or c<i and i>c | A⊥ |

| | |
|---|---|
| or c<i and i>c | LEAA⊥ |
| or c<i and i>c | orEAA⊥ |
| c<i and i>c | EAA⊥ |
| c<i and i>c | VBAA⊥ |
| c<i and i>c | cBAA⊥ |
| <i and i>c | BAA⊥ |
| <i and i>c | RVAA⊥ |
| <i and i>c | <VAA⊥ |
| i and i>c | VAA⊥ |
| i and i>c | iAA⊥ |
| and i>c | AA⊥ |
| and i>c | LEAA⊥ |
| and i>c | andEAA⊥ |
| i>c | EAA⊥ |
| i>c | VBAA⊥ |
| i>c | iBAA⊥ |
| >c | BAA⊥ |
| >c | RVAA⊥ |
| >c | >VAA⊥ |
| c | VAA⊥ |
| c⊥ | cAA⊥ |
| ⊥ | AA⊥ |
| ⊥ | A⊥ |
| ⊥ | ⊥ |
| | |

### *5.3 Задания*

1. Вычислите значения функций FIRST (для каждого правила) и FOLLOW (для каждого нетерминального символа) для следующих грамматик. Определите, является ли грамматика LL(1) грамматикой.

A) $S \rightarrow xSx$

$S \rightarrow x$

B) S→xSyS

S→ySxS

S→ε

2. Проанализируйте приведенные ниже грамматики. В случае если грамматика не является LL(1) грамматикой, выполните необходимые эквивалентные преобразования (устранение левой рекурсии, левую факторизацию). В случае необходимости, введите символ, ограничивающий входную цепочку, и постройте расширенную грамматику. С использованием полученной грамматики постройте LL(1) распознаватель и выполните с его помощью анализ указанных цепочек. В случае если анализируемая цепочка не принадлежит языку, сформируйте сообщение об ошибке с указанием способа ее исправления.

A) Грамматика: S→xSy

S→z

Цепочки: a) xzyy

b) xxxzyyy

c) xxyy

B) Грамматика: S→xSy

L→x

Цепочки: a) xxxyy

b) xxxyyy

c) xyxy

d) xzy

C) Грамматика: S→aBd

B→bBcc|ε

Цепочки: a) abbccccd

b) abacd

c) abcad

d) abcc

D) Грамматика: S→ZSS|i

Z→-|+

Цепочки: a) +i++iii

b) +ic

c) ++ii

E) Грамматика: S→SSZ|c

Z→-|+

Цепочки: a) cc+cc+-

b) c+

c) cc

F) Грамматика: S→ar L

L→[V]L|ε

V→i|c

Цепочки: a) ar[i][c]

b) ar[i)[c]

c) ar[]

G) Грамматика: S→ar L

L→L[V]|ε

V→i|S

Цепочки: a) ar[i][ai]

b) ar[i][ar[ar[i][i]]]

H) Грамматика: S→aZS|a|(S)

Z→<|<=

Цепочки: a) a<=(a<a<=a)

b) a<a=a

c) a<=

# 6. Восходящий синтаксический анализ КС-языков. Грамматики предшествования

## *6.1 Краткие теоретические сведения*

Вероятно, самым простым и наглядным способом восходящего анализа контекстно-свободных языков является их анализ на основе отношений предшествования. Отношения предшествования устанавливаются между символами грамматики языка на этапе разработки анализатора по правилам грамматики языка. На этапе синтаксического анализа отношения предшествования позволяют производить последовательное выделение основ и их свертку, обеспечивая восходящее распознавание входной цепочки. Было предложено несколько методов анализа на основе отношений предшествования, однако мы рассмотрим наиболее известный из них и применяемый к достаточно узкому классу контекстно-свободных грамматик – классу грамматик простого предшествования.

Прежде чем перейти к определению отношений предшествования, введем две функции: Left(A) и Right(A).

Пусть задана грамматика $(S, V_N, V_T, R)$. *Функция Left(A)* для любого нетерминального символа A ($A \in V_N$) грамматики возвращает множество «левых» символов – символов, с которых могут начинаться цепочки, выводимые из A:

$Left(A)=\{x \mid A \Rightarrow^* x\alpha,\ x \in (V_T \cup V_N),\ \alpha \in (V_T \cup V_N)^*\}$

*Функция Right(A)* для любого нетерминального символа A ($A \in V_N$) грамматики возвращает множество «правых» символов – символов, которыми могут заканчиваться цепочки, выводимые из A:

$Right(A)=\{x \mid A \Rightarrow^* \alpha x,\ x \in (V_T \cup V_N),\ \alpha \in (V_T \cup V_N)^*\}$

Обратите внимание на отличия между множествами Left(A) и $First_1(A)$. Первое из них включает как терминальные, так и нетерминальные символы, с которых будут начинаться цепочки, выводимые из A. Второе – только односимвольные терминальные цепочки, выводимые из A, и, возможно, пустую цепочку (в случае, если возможно $A \Rightarrow^* \varepsilon$).

Рекурсивные алгоритмы построения множеств Left и Right достаточно просты:

Пусть $A \in V_N$, тогда Left(A) вычисляется следующим образом:

1. Инициализируем $Left(A) := \varnothing$.

2. **Для каждого** правила $(A \to x\alpha) \in R$ **выполняем**:

2.1 $Left(A) := Left(A) \cup \{x\}$, $x \in (V_T \cup V_N)$, $\alpha \in (V_T \cup V_N)^*$

2.2 **Если** $x \in V_N$, **то** $Left(A) := Left(A) \cup Left(x)$.

Аналогично для Right(A) имеем:

1. Инициализируем Right(A) := ∅.

2. **Для каждого** правила $(A \to \alpha x) \in R$ **выполняем**:

2.1. $Right(A) := Right(A) \cup \{x\}$, $x \in (V_T \cup V_N)$, $\alpha \in (V_T \cup V_N)^*$

2.2. **Если** $x \in V_N$, **то** $Right(A) := Right(A) \cup Right(x)$.

Рассмотрим теперь, какие отношения и как устанавливаются в грамматиках простого предшествования. Пусть x и y – два символа грамматики: $x,y \in (V_T \cup V_N)$.

1. Между символами x и y *отношения предшествования равенства* ($x \doteq y$) устанавливается тогда и только тогда, когда

$(A \to \alpha x y \beta) \in R$, $A \in (V_T \cup V_N)$, $\alpha,\beta \in (V_T \cup V_N)^*$

То есть отношения предшествования равенства устанавливается в том случае, когда символы x и y стоят рядом в каком-либо правиле вывода. Нахождение таких отношений не представляет никаких затруднений.

2. Между символами x и y *отношения предшествования меньше* ($x \lessdot y$) устанавливается тогда и только тогда, когда

$(A \to \alpha x B \beta) \in R$ и $B \Rightarrow^+ y\varphi$, $A,B \in (V_T \cup V_N)$, $\alpha,\beta,\varphi \in (V_T \cup V_N)^*$

То есть отношения предшествования меньше устанавливается в том случае, когда x уже выведено, а y встанет непосредственно справа от x на следующих шагах вывода. Для поиска таких отношений просматривают все правила вида $A \to \alpha x B \beta$ и устанавливают отношения предшествования $x \lessdot y$ в случае, если $y \in Left(B)$.

3. Между символами x и y *отношения предшествования больше* ($x \gtrdot y$) устанавливается тогда и только тогда, когда:

$(A \to \alpha B C \beta) \in R$ и $B \Rightarrow^+ \varphi x$, $C \Rightarrow^* y\psi$, $A,B \in (V_T \cup V_N)$, $\alpha,\beta,\varphi,\psi \in (V_T \cup V_N)^*$

Последнее условие эквивалентно двум следующим:

$(A \to \alpha B y \beta) \in R$ и $B \Rightarrow^+ \varphi x$, $A,B \in (V_T \cup V_N)$, $\alpha,\beta,\varphi \in (V_T \cup V_N)^*$

$(A \to \alpha B C \beta) \in R$ и $B \Rightarrow^+ \varphi x$, $C \Rightarrow^+ y\psi$, $A,B \in (V_T \cup V_N)$, $\alpha,\beta,\varphi,\psi \in (V_T \cup V_N)^*$

Первое из приведенных условий говорит о том, что $x \gtrdot y$ в том случае, когда y уже выведено, а x встанет непосредственно слева от y на следующих шагах вывода. К сожалению, в том случае, когда y является нетерминальным символом только первого условия недостаточно, чтобы обнаружить правую границу для свертки. Дело в том, что обработка цепочки производится слева-направо, и подцепочка, стоящая справа от x, не может быть свернута к нетерминальному символу y, так как ее обработка еще не проводилась. По этой причине отношения предшествования $x \gtrdot y$ устанавливается и тогда,

когда x и y являются соседними символами, выводимыми на разных шагах вывода из сентенциальной формы …BC…

Для построения отношений x•>y просматривают все правила вида A→αByβ и устанавливают отношения предшествования x•>y в случае, если x∈Right(B). Затем просматривают все правила вида A→αBCβ и устанавливают отношения x•>y в случае, если x∈Right(B), y∈Left(C).

После построения отношений предшествования для некоторой грамматики можно определить, является грамматика грамматикой простого предшествования. Дадим определение такому классу грамматик.

Контекстно-свободная грамматика является *грамматикой простого предшествования*, если:

1) в грамматике не содержится аннулирующих правил,

2) между любыми двумя символами грамматики существует не более одного отношения предшествования,

3) в грамматике нет правил с одинаковыми правыми частями.

Анализ входной цепочки с использованием отношений предшествования выполняется следующим образом. Входная цепочка просматривается слева направо и между каждой парой просматриваемых символов устанавливаются отношения предшествования. Так продолжается до тех пор, пока не будет найдена подцепочка, «заключенная» в отношения предшествования меньше (левая граница подцепочки) и больше (правая граница подцепочки), между всеми символами которой установлены отношения предшествования равенства. Выделенная подцепочка является основой для свертки и на следующем шаге анализа заменяется на некоторый нетерминал в соответствие с правилом грамматики, найденным по выделенной основе для свертки. Анализ продолжается до тех пор, пока исходная цепочка не будет свернута до начального символа грамматики.

Во время анализа возможно возникновение следующих ошибок, свидетельствующих о том, что цепочка не принадлежит языку, порождаемому грамматикой:

- при просмотре цепочки между парой соседних символов не установлено отношения предшествования,

- при просмотре цепочки не найдена основа для свертки,

- при просмотре цепочки основа для свертки найдена, но ей не соответствует ни одно из правил грамматики (правые части правил не совпадают с выделенной основой).

## *6.2 Пример*

**Задание.** Проанализировать приведенную ниже грамматику. В случае если грамматика не является грамматикой простого предшествования, преобразовать грамматику. Построить матрицу предшествования и выполнить анализ указанной цепочки.

Грамматика: S→if condition then O E endif

O→S|operator

E→else O|ε

Цепочка: if condition then

if condition then operator

endif

else

if condition then operator

endif

endif

**Решение.**

Для удобства изложения введем следующие обозначения для цепочек терминальных символов исходной грамматики:

i – ‘if condition then’,

o – ‘operator’,

e – ‘else’,

f – ‘endif’.

С использованием указанных обозначений исходная грамматика примет следующий вид:

S→iOEf

O→S|o

E→eO|ε

Исходная цепочка будет выглядеть следующим образом: iiofeioff

Очевидно, грамматика не является грамматикой простого предшествования, так как содержит аннулирующее правило E→ε. Так как нетерминал E содержится в правой части только одного правила S→iOEf, то к грамматике достаточно добавить правило S→iOf, после чего правило E→ε можно удалить. После устранения аннулирующего правила грамматика будет выглядеть следующим образом

S→iOEf| iOf

O→S|o

E→eO

Формально, грамматика является обратимой и не содержит аннулирующих правил. Поэтому в том случае, если для любых двух символов будет существовать не более одного отношения предшествования, грамматика будет являться грамматикой простого предшествования.

Построим отношения предшествования. Для этого вначале сформируем множества левых и правых символов для всех нетерминалов грамматики.

Так как самыми левыми и самыми правыми символами для обоих правил нетерминала S являются терминальные символы i и f соответственно, то очевидно

Left(S)={i}, Right(S)={f}.

Далее, так как оба правила нетерминала O содержат по одному символу (S и o), то оба символа входят в множества как левых, так и правых символов. Кроме того, символ S является нетерминальным, поэтому множества левых и правых символов для O необходимо дополнить элементами множеств Left(S) и Right(S) соответственно.

Left (O)={S,o}∪ Left(S)={S,o,i}, Right(O)={S,o}∪Right(S)={S,o,f}.

Для нетерминала E имеем

Left(E)={e}, Right(E)={O}∪Right(O)= {O}∪{S,o,f}={O,S,o,f}.

Для удобства сформированные множества представим в виде таблицы

| **N** | Left**(N)** | **Right(N)** |
|---|---|---|
| **S** | i | f |
| **O** | S,o,i | S,o,f |
| **E** | e | O,S,o,f |

Построим отношения предшествования для первого правила грамматики S→iOEf.

Отношение равенства s1≐s2 устанавливаются между соседними символами s1 и s2, если они стоят рядом в одном правиле вывода (s1 непосредственно перед s2). Поэтому для первого правила:

i≐O, O≐E, E≐f.

Далее, отношение предшествования меньше s1<•s2 устанавливается между символами s1 и s2, если s1 стоит в правиле непосредственно перед некоторым нетерминалом A, а символ s2 является левым символом для нетерминала A.

В нашем случае отношение меньше будет установлено:

- между i и всеми левыми символами нетерминала O

i<•S, i<•o, i<•i;

- между O и всеми левыми символами нетерминала E

O<•e.

Наконец, отношение предшествования больше s1•>s2 устанавливается между символами s1 и s2 в двух случаях:

1) если некоторый нетерминал A стоит в правиле непосредственно перед символом s2, а символ s1 является правым символом для этого A,
2) если некоторый нетерминал A стоит в правиле непосредственно перед другим нетерминалом B, символ s1 является правым символом для нетерминала A, а символ s2 - левым символом для нетерминала B.

В нашем случае отношение больше будет установлено:

- между всеми правыми символами нетерминала O и символом E

S•>E, o•>E, f•>E;

- между всеми правыми символами нетерминала O и всеми левыми символами нетерминала E

S•>e, o•>e, f•>e;

- между всеми правыми символами нетерминала E и символом f

O•>f, S•>f, o•>f, f•>f.

Далее для второго правила грамматики S→iOf получим

i≐O, O≐f;

i<•S, i<•o, i<•i;

S•>f, o•>f, f•>f.

Нетрудно видеть, что для рассматриваемой грамматики имеет место конфликт. Из первого правила грамматики следует O•>f, но из второго правила следует, что O≐f. Это означает, что рассматриваемая грамматика не является грамматикой простого предшествования.

Тем не менее, в данном случае наличие конфликта не является неразрешимой проблемой. Чтобы понять суть проблемы проанализируем создавшуюся ситуацию. Отношение больше в данном случае возникает из-за того, что O оказывается правым символом нетерминала E, который в первом правиле стоит непосредственно перед f. Отношение равенства возникает из-за того, что O и f стоят рядом во втором правиле вывода. Для того чтобы обойти создавшийся конфликт достаточно избавиться от отношения предшествования равенства путем изменения правил на следующие

S→iO’Ef| iO’f

и введения дополнительного правила O’→O.

Внесение таких несложных изменений позволяет заменить отношение O≐f на O'≐f, что является достаточным для разрешения конфликтной ситуации.

Таким образом, модифицированная грамматика принимает следующий вид:

S→iO'Ef| iO'f

O'→O

O→S|o

E→eO

Сформируем множества левых и правых символов (представлены в виде таблицы)

| **N** | Left**(N)** | **Right(N)** |
|---|---|---|
| **S** | i | f |
| **O'** | O,S,o,i | O,S,o,f |
| **O** | S,o,i | S,o,f |
| **E** | e | O,S,o,f |

Отношения предшествования для модифицированной грамматики представлены ниже в виде матрицы предшествования

| | **S** | **i** | **O** | **E** | **f** | **O'** | **o** | **e** |
|---|---|---|---|---|---|---|---|---|
| **S** | | | | ●> | ●> | | | ●> |
| **i** | <● | <● | <● | | | ≐ | <● | |
| **O** | | | | ●> | ●> | | | ●> |
| **E** | | | | | ≐ | | | |
| **f** | | | | ●> | ●> | | | ●> |
| **O'** | | | | ≐ | ≐ | | | <● |
| **o** | | | | ●> | ●> | | | ●> |
| **e** | <● | <● | ≐ | | | | <● | |

Несложно убедиться, что для любых двух символов модифицированной грамматики существует не более одного отношения предшествования. Следовательно, грамматика является грамматикой простого предшествования.

С использованием построенных отношений предшествования выполним анализ заданной цепочки (на каждом шаге анализа выделенные основы для свертки показаны подчеркнутым жирным шрифтом).

i <● i <● **o** ●> f e i o f f

i <● i <● **O** ●> f e i o f f

i <● **i ≐ O' ≐ f** ●> e i o f f

i <● **S** ●> e i o f f

i <• **O** •> e i o f f

i ≐ O’ <• e <• i <• **o** •> f f

i ≐ O’ <• e <• i <• **O** •> f f

i ≐ O’ <• e <• **i ≐ O’ ≐ f** •> f

i ≐ O’ <• e <• **S** •> f

i ≐ O’ <• **e ≐ O** •> f

**i ≐ O’ ≐ E ≐ f**

**S**

Итак, цепочку удалось свернуть до начального символа грамматики, следовательно, цепочка принадлежит языку, порождаемому рассматриваемой грамматикой.

**Примечание 1**

В рассмотренной задаче имела место ситуация, при которой между двумя символами грамматики установлено более одного отношения предшествования. Такая ситуация возникает на практике довольно часто, в частности, из-за наличия в правилах грамматики рекурсии.

Пусть имеется правило вида A→…aB… и леворекурсивное правило B→B… В этом случае нетрудно убедиться, что имеют место отношения a≐B и a<•B. Избежать такого конфликта и получить эквивалентную грамматику позволяет прием, называемый стратификацией. Суть его состоит в замене правила A→…aB… на правило A→…aB’… и добавлении нового правила B’→B.

В случае правосторонней рекурсии имеем правила A→…Ba… и B→…B и отношения B≐a и B•>a. Заменяем A→…Ba… на A→…B’a… и добавляем B’→B.

Для примера рассмотрим простейшую грамматику

S→aA

A→Ab|b

Нетрудно убедиться, что имеют место отношения a≐A и a<•A. После проведенной модификации грамматика принимает вид

S→aA’

A’→A

A→Ab|b

Отношения предшествования для модифицированной грамматики представлены ниже:

a≐A’, A≐b; a<•A, a<•b; b•>b.

Как видно, модифицированная грамматика относится к классу грамматик предшествования.

**Примечание 2**

Цепочки языка, порождаемые рассмотренной выше простой грамматикой, тем не менее, не могут быть непосредственно проанализированы с использованием отношений предшествования. Действительно, даже анализ кратчайшей цепочки ab, выводимой из данной грамматики за 3 шага не может быть выполнен, так как основа для свертки не может быть выделена: a<•b.

Решение этой проблемы весьма просто: вводится символ, ограничивающий входную цепочку (например, ⊥) и грамматика расширяется правилом для нового начального символа грамматики (S'→⊥S⊥).

В частности, рассмотренная выше грамматика принимает вид

S'→⊥S⊥ S→aA'

A'→A A→Ab|b

Отношения предшествования для расширенной грамматики следующие:

⊥≐S, S≐⊥, a≐A', A≐b;

⊥<•a, a<•A, a<•b;

A'•>⊥, A•>⊥, b•>⊥, b•>b.

Анализ цепочки abb выглядит следующим образом:

⊥ <• a <• **b** •> b⊥

⊥ <• a <• **A ≐ b** •> ⊥

⊥ <• **a ≐ A'** •> ⊥

**⊥ ≐ S ≐ ⊥**

**S'**

Следует, однако, помнить, что класс грамматик предшествования достаточно узок и с использованием стратификации или других преобразований не всегда удается избавиться от конфликтов.

## *6.3 Задачи*

1. Постройте множества левых и правых символов (для нетерминалов грамматики) и отношения предшествования для приведенной ниже грамматики. Определите, является ли рассматриваемая грамматика грамматикой предшествования.

A) S→xSy|A

A→yAx|yx

B) S→xSyS|ySxS|x|y

C) S→xSy|A

A→yAx|yx

2. Проанализируйте приведенные ниже грамматики. В случае если грамматика не является грамматикой простого предшествования, выполните необходимые преобразования (постройте обратимую грамматику, введите символ, ограничивающий входную цепочку, и постройте расширенную грамматику, выполните стратификацию). С использованием полученной грамматики постройте матрицу предшествования и выполните анализ указанных цепочек.

A) Грамматика: S→[L]
L→L,i|i
Цепочка: [i,i,i]

B) Грамматика: S→xSx|xAy
A→Az|ε
Цепочка: xxzzzyy

C) Грамматика: S→iL
L→[i]L|[i]
Цепочка: i[i][i]

D) Грамматика: S→(L)
L→C|C,L|S|S,L
Цепочка: (10,((20,30),40,50)

E) Грамматика: S→st{V};
V→T;|VT;
T→tL
L→i|L,i
Цепочка: st{t i,i; t i;};

F) Грамматика: S→S+T|T
T→T*V|V
V→c|(S)
Цепочка: c+c*(c+c)*c

G) Грамматика: S→SS+|i
Цепочка: ii+ii++

H) Грамматика: S→+SS|I
Цепочка: ++i+iii

# 7. Восходящий синтаксический анализ КС-языков. LR(k) грамматики

## *7.1 Краткие теоретические сведения*

Наиболее широким классом контекстно-свободных грамматик, допускающим восходящий синтаксический анализ, являются *LR(k) грамматики*. Распознаватели, построенные по таким грамматикам, при просмотре входной цепочки слева направо выполняют восстановление ее правого канонического вывода. В настоящем разделе мы рассмотрим только самый простой подкласс таких грамматик - LR(0) грамматики. Дадим вначале необходимые определения.

Пусть задана контекстно-свободная грамматика $G=(S, V_N, V_T, R)$. *Ситуацией* будем называть продукцию грамматики G, в правой части которой указана точка просмотра сентенциальной формы: .

$<A\rightarrow\alpha\bullet\beta>$, где $A\in V_N$, $\alpha,\beta\in(V_N\cup V_T)^*$, «•» - обозначение позиции просмотра

*Ситуационным множеством* будем называть множество ситуаций, указанного выше вида.

Первым шагом при построении LR(0) анализатора является построение ситуационных множеств для грамматики. Такое построение производится следующим образом:

1) **Построение начального ситуационного множества.**

Пусть для начального символа грамматики имеется следующий набор правил:

$S\rightarrow\alpha_1|\alpha_2|\ldots|\alpha_N$, где $\alpha_N\in(V_N\cup V_T)^*$

Тогда в начальное ситуационное множество $M_0$ включаются ситуации, образованные правилами для нетерминального символа S, в которых точка просмотра стоит непосредственно перед правыми частями правил:

$M_0:=\{<S\rightarrow\bullet\alpha_1>, <S\rightarrow\bullet\alpha_2>, \ldots, <S\rightarrow\bullet\alpha_N>\}$

Строим замыкание (Closure) рассматриваемого ситуационного множества с использованием шага 2.

2) **Построение замыкания ситуационного множества (операция Closure).**

В том случае, если в рассматриваемое ситуационное множество $M_i$ входит ситуация $<A\rightarrow\alpha\bullet B\beta>$, в которой точка просмотра стоит перед нетерминальным символом B, добавляем в рассматриваемое ситуационное множество ситуации, образованные правилами для нетерминального символа B, в которых точка просмотра стоит непосредственно перед правыми частями правил:

**Если** $<A\rightarrow\alpha\bullet B\beta>\in M_i$ и $B\in V_N$, **то** $M_i:=M_i\cup\{<B\rightarrow\bullet\beta_j> \mid B\rightarrow\bullet\beta_j\in R\}$

Повторяем шаг 2 до тех пор, пока множество не перестанет изменяться.

После этого переходим к шагу 3.

3) **Построение переходов в новые ситуационное множества (операция Goto)**. Для рассматриваемого ситуационного множества $M_i$, строим переходы в новые ситуационные множества. В том случае, если в рассматриваемое ситуационное множество $M_i$ входит ситуация $<A \rightarrow \alpha \bullet x \beta>$, в которой точка просмотра стоит перед некоторым символом x (как терминальным, так и нетерминальным), образуем новое ситуационное множество $M_j$ путем перехода из множества $M_i$ по символу x. Во вновь образованное множество добавляем ситуацию $<A \rightarrow \alpha x \bullet \beta>$, в которой точка просмотра перенесена через символ x по сравнению с исходной ситуацией:

**Если** $<A \rightarrow \alpha \bullet x \beta> \in R$, **то** $M_j := goto(M_i, x) = \{<A \rightarrow \alpha x \bullet \beta>\}$

В том случае, если во вновь образованное множество входит ситуация, в которой точка просмотра стоит за правой частью правила грамматики ($<A \rightarrow \gamma \bullet>$), то такая ситуация называется терминальной (заключительной). Для всех образованных на шаге 3 множеств, за исключением тех, которые содержат терминальные ситуации, выполняем шаг 2.

После построения всех ситуационных множеств, можно перейти к построению LR(0)-распознавателя. Вначале, однако, дадим определение LR(0) грамматики.

Контекстно-свободная грамматика является *LR(0) грамматикой*, если любое ситуационное множество, содержащее терминальную ситуацию, не содержит никаких других ситуаций, кроме терминальной.

Собственно LR(0) распознаватель представляет собой конечный автомат $(\Sigma, Q, q_0, \delta, F)$. Алфавит $(\Sigma)$ автомата представляет собой объединение множеств терминальныи и нетерминальных символов исходной грамматики $(V_N \cup V_T)$. Множество состояний (Q) автомата определяется сформированными ситуационными множествами $\{M_0, M_1, \ldots M_K\}$. Начальное состояние $(q_0)$ автомата определяется начальным ситуационным множеством $M_0$. Множество допускающих состояний (F) автомата определяется ситуационными множествами, содержащими терминальные ситуации. Функция переходов $(\delta)$ автомата однозначно связана с операцией перехода в новое ситуационное множество: $\delta(M_i, x) = Goto(M_i, x)$.

Распознавание входной цепочки с использованием построенного автомата выполняется следующим образом. Входная цепочка подается на вход автомата и под ее управлением автомат проделывает последовательность шагов. В том случае, если в результате автомат приходит в заключительное состояние, то основа для свертки считается выделенной, а соответствующая терминальная ситуация показывает, в соответствии с каким правилом грамматики должна быть произведена сверка. После выполнения сверки (замены

символов цепочки, совпадающих с правой частью правила на нетерминальный символ, стоящий в левой части правила) цепочка подается на вход автомата заново и так далее до тех пор, пока исходная цепочка не будет свернута к начальному символу грамматики или не будет обнаружена ошибка при анализе цепочки.

Об обнаружении ошибки свидетельствует отсутствие перехода из некоторого состояния автомата по текущему символу входной цепочки. В этом случае наличие переходов по другим символам может указать на способ исправления ошибки во входной цепочке. Ситуация, при которой достигнут конец анализируемой цепочки, а конечное состояние автомата не было достигнуто, свидетельствует о преждевременном конце (обрыве) входной цепочки. Наоборот, ситуация при которой свертка до начального символа уже произведена, а в цепочке остались непрочитанные символы, свидетельствует о лишних символах во входной цепочке.

## *7.2 Пример*

**Задание.** Построить LR(0) анализатор КС-грамматики, порождающей цепочки вида $x^n y^m x^m y^n$, где $n \geq 0$, $m > 0$.

**Решение**.

Грамматика, порождающая цепочки заданного вида может быть составлена следующим образом.

$S \rightarrow xSy|A$

$A \rightarrow yAx|yx$

Перейдем от построенной грамматики к расширенной правилом $S' \rightarrow S$ грамматике:

(1) $S' \rightarrow S$

(2) $S \rightarrow xSy|A$

(3) $A \rightarrow yAx|yx$

Такой переход осуществляется для того, чтобы при свертке входных цепочек мы могли однозначно принять решение о завершении процесса анализа цепочки. К этому вопросу вернемся позднее.

Построим ситуационные множества для расширенной грамматики.

Для правила (1) имеем:

$M_0$:={ <S'→•S> }.

Так как точка просмотра в единственной ситуации множества стоит перед нетерминальным символом, строим замыкание множества с использованием функции Closure:

$M_0$:=Closure($M_0$)={ <S'→•S>; <S→•xSy >; <S→•A>; <A→•yAx>; < A→•yx> }.

Далее с использованием функции goto строим следующие ситуационные множества, переходы в которые из $M_0$ будут осуществляться по тем символам грамматики, которые стоят в ситуациях $M_0$ непосредственно после точек просмотра.

$M_1$:=Goto($M_0$,S) = { <S'→S•> }

$M_2$:=Goto($M_0$,x) = { < S→x•Sy> }

$M_2$:=Closure($M_2$) = { < S→x•Sy> <S→•xSy >; <S→•A>; <A→•yAx>; < A→•yx> }

$M_3$:= Goto ($M_0$,A) = { <S→A•> }

$M_4$:= Goto ($M_0$,y) = { <A→y•Ax>; < A→y•x> }

$M_4$:= Closure ($M_4$) = { <A→y•Ax>; < A→y•x>; <A→•yAx>; < A→•yx> }

Обратим внимание на то, что множества $M_1$,$M_3$ (подчеркнуты) содержат по единственной заключительной ситуации. Точки просмотра в таких ситуациях стоят в конце правила.

Отметим также и то, что при построении ситуационных множеств $M_2$ и $M_4$ мы строили замыкания с использованием функции Closure, так как точки просмотра в ситуациях < S→x•Sy> и <A→y•Ax> оказывались непосредственно перед нетерминальными символами грамматики.

Ниже по аналогии построим остальные ситуационные множества.

$M_5$:= Goto ($M_2$,S) = { <S→xS•y> }

$M_6$:= Goto ($M_2$,x) = { <S→x•Sy> }

$M_6$:= Closure ($M_6$) = { < S→x•Sy> <S→•xSy >; <S→•A>; <A→•yAx>; < A→•yx> } = $M_2$

Заметим, что построенное ситуационное множество $M_6$ совпадает с построенным ранее $M_2$. Вообще говоря, замыкание этого множества можно было и не строить, так как результат Goto полностью совпал с результатом Goto($M_0$,x). В дальнейшем, при возникновении подобных ситуаций мы не будем вводить для таких множеств новых обозначений.

Goto ($M_2$,A) = { <S→A•> } ... = $M_3$

Goto ($M_2$,y) = { <A→y•Ax>; < A→y•x> } ... = $M_4$

$M_7$:= Goto ($M_4$,A) = { <A→yA•x> }

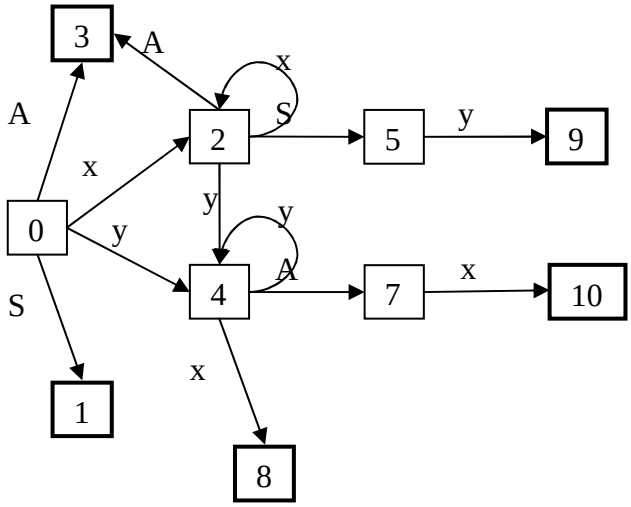
$M_8$:= Goto ($M_4$,x) = { < A→yx•> }

Goto ($M_4$,y) = { <A→y•Ax>; < A→y•x> } ... = $M_4$

$M_9$:=Goto ($M_5$,y) = { <S→xSy•> }

$M_{10}$:= Goto ($M_7$,x) = { <A→yAx•> }

После того, как все ситуационные множества построены, перейдем к построению конечного автомата, реализующего LR(0) распознаватель. На следующем рисунке показан граф переходов распознавателя.




С использованием полученного распознавателя выполним анализ цепочки xxyyyxxxyy.

| **Состояние распознавателя** | **Цепочка** | **Стек** |
|---|---|---|
| 0 | **x** x y y y x x x y y | |
| 2 | x **x** y y y x x x y y | x |
| 2 | x x **y** y y x x x y y | x x |
| 4 | x x y **y** y x x x y y | x x y |
| 4 | x x y y **y** x x x y y | |
| 4 | x x y y y **x** x x y y | |
| 8 | | |

Переход конечного автомата в заключительное состояние означает, что достигнут конец основы для свертки, которая должна быть произведена в соответствие с правилом A→yx, соответствующим состоянию 8. То есть должна быть произведена замена выделенной подцепочки

x x y y **y x** x x y y

на нетерминальный символ A. После замены имеем:

x x y y A x x y y

Аналогичным образом произведем последовательно нахождение и свертку основ вплоть до конечного символа.

| Входная цепочка | Последовательность состояний | Выделенная основа |
|---|---|---|
| x x y y A x x y y | 0-2-2-4-4-7-10 | x x y **y A x** x y y |
| x x y A x y y | 0-2-2-4-7-10 | x x **y A x** y y |
| x x A y y | 0-2-2-3 | x x **A** y y |
| x x S y y | 0-2-2-5-9 | x **x S y** y |
| x S y | 0-2-5-9 | **x S y** |
| S | 0-1 | **S** |

После выполнения последней редукции S'→S получим начальный символ модифицированной грамматики S', что указывает на принадлежность входной цепочки языку.

## *7.3 Задачи*

1. Постройте ситуационные множества и LR(0) распознаватель для приведенной ниже грамматики (в случае необходимости, введите символ, ограничивающий входную цепочку, и постройте расширенную грамматику). С использованием построенного распознавателя выполните анализ указанных цепочек. Если анализируемая цепочка не принадлежит языку, укажите возможную причину и способ формирования информативного сообщения об ошибке.

A) Грамматика: S→[S]|x

Цепочки: a) [[x]

b) [[x]]

c) [[x)]

B) Грамматика: S→a[L]|a[]

L→L,P|P

P→i

Цепочки: a) a[]

b) a[i]

c) a[i,i]

d) a[i,]

e) a[]i

C) Грамматика: S→f(L)|f()

L→L,P|P

P→S|c

Цепочки: a) f(f())

b) f(f(c,f()))

c) f(c,i)

d) f(f(c)

D) Грамматика: S→SA|A

A→op;|{S}

Цепочки: a) op;{op;{op;}op;}

b)

c)

E) Грамматика: S→S+V|S-V|V

V→i|c

Цепочки: a) i+c-i+i-c

b)

c)

F) Грамматика: T→T+S|T

T→T*i|i

Цепочки: a) i+i*i+i*i

b)

c)

G) Грамматика: S→SS+|SS*|c

Цепочки: a) ccc+*c+

b)

c)

H) Грамматика: S→AB

A→xAy|xy

B→Bz|z

Цепочки: a) xxyyzz

b) xxyz

c) xy

Для задания под буквой B,E подумайте, возможно ли сформировать эквивалентную автоматную грамматику? Какой анализатор в этом случае следует предпочесть?

# 8. Польская инверсная запись

## *8.1 Краткие теоретические сведения*

Наиболее известной из форм внутреннего представления программ является *польская инверсная запись (ПОЛИЗ)*, известная также как *постфиксная нотация*. В отличие от обычной инфиксной формы записи в постфиксной нотации операции записываются после аргументов, а не между ними. Например, операции a + b и a := b, представленные в традиционной инфиксной форме, в постфиксной нотации примут вид: a b + и a b :=.

Операции в выражениях, представленных в постфиксной форме записи, выполняются в порядке записи слева-направо, приоритеты и ассоциативность операций не принимаются во внимание. Не используются в постфиксной форме записи и скобки.

Для вычисления вручную выражения, представленного в постфиксной форме записи, оно читается слева-направо. Как только в выражении встречается операция, она выполняется над двумя стоящими непосредственно перед ней операндами, после чего операнды и операция вычеркиваются из выражения и на их место вписывается результат выполнения операции. После этого выражение вычисляется дальше по тому же правилу, пока не будет получено единственное значение - результат вычисления выражения.

В качестве примера рассмотрим вычисление выражения (2+3)*7+4/2. Соответствующая ему польская инверсная запись имеет вид: 2 3 + 7 * 4 2 / +.

**2 3 +** 7 * 4 2 / +

**5 7 *** 4 2 / +

35 **4 2 /** +

**35 2 +**

37

Составить вручную ПОЛИЗ по инфиксной записи для небольших выражений не составляет большого труда. Для этого необходимо помнить, что операции в ПОЛИЗе всегда следуют в том порядке, в котором они должны выполняться, а операнды – в том же порядке, в каком они следовали в инфиксной записи. При этом операторы должны располагаться в ПОЛИЗе сразу же за своими операндами.

Следует отметить, что польская инверсная запись зачастую короче инфиксной формы записи. Это дает возможность уменьшить объём программ по сравнению с традиционной формой записи, что немаловажно для тех вычислительных устройств, где имеются жёсткие требования по использованию памяти.

Следует отметить, что с использованием постфиксной формы записи возможно описание не только арифметических операций, но и любых других операций, в частности операций управления потоком вычислений в программе (см. таблицу 3.1).

Таблица 3.1 – Операции в постфиксной форме записи

| **Операция** | **Описание** | **Формат** |
|---|---|---|
| JMP | Команда безусловного перехода | унарная |
| JZ | Команда условного перехода по лжи (если значение первого аргумента ложно - имеет нулевое значение) | бинарная |
| := | Операция присваивания | бинарная |
| + | Операция сложения | бинарная |
| - | Операция вычитания | бинарная |
| * | Операция умножения | бинарная |
| / | Операция деления | бинарная |
| NOT | Операция логического «не» | унарная |
| AND | Операция логического «и» | бинарная |
| OR | Операция логического «или» | бинарная |
| == | Операция сравнения на равенство | бинарная |
| != | Операция сравнения на неравенство | бинарная |
| < | Операция сравнения «строго меньше» | бинарная |
| <= | Операция сравнения «меньше или равно» | бинарная |
| > | Операция сравнения «строго больше» | бинарная |
| >= | Операция сравнения «больше или равно» | бинарная |
| << | Ввод значения из входного потока | унарная |
| >> | Вывод значения во входной поток | унарная |

Этот факт позволяет использовать постфиксную форму записи в качестве внутренней формы представления программы. Рассмотрим, как в ПОЛИЗе будут выглядеть традиционные управляющие конструкции языков высокого уровня.

Условный оператор, имеющий вид

```
if (<условие>) <оператор1> else <оператор2>;
```

в ПОЛИЗе может быть представлен выражением:

```
<условие> <адр7> JZ   <оператор1> <адр8> JMP  <оператор2> …
адр1      адр2   адр3 адр4        адр5   адр6 адр7        адр8
```

Оператор цикла с предусловием, имеющий вид

```
while (<условие>) <оператор1>;
```

в ПОЛИЗе выглядит следующим образом:

```
<условие> <адр7> JZ   <оператор1> <адр1> JMP  …
адр1      адр2   адр3 адр4        адр5   адр6 адр7
```

Оператор цикла с постусловием, имеющий вид

```
do <оператор1> while (<условие>);
```

в ПОЛИЗе выглядит следующим образом:

```
<оператор1> <условие> NOT  <адр1> JZ   …
адр1        адр2      адр3 адр4   адр5 адр6
```

Для представления цикла

```
for (<выражение1>; <выражение2>; <выражение3>) <оператор>
```

достаточно вспомнить, что этот цикл эквивалентен циклу с предусловием:

```
<выражение1>;
while (<выражение2>) {
    <оператор>;
    <выражение3>;
}
```

Особое внимание следует уделить операции обращения по индексу. В том случае, когда допускается использование только одномерных массивов, такая операция в ПОЛИЗе может иметь вид

```
<имя_массива> <индекс> [
```

Если допустимо использование многомерных массивов операция принимает вид:

```
<имя_массива> <индекс1><индекс2>…<индексN> N [
```

В качестве примера приведем следующий фрагмент программы на языке С:

```
for (i=0; i<10; i++) {
    if (i<5)&&(j>0)  a[i] = j;
    else  a[i]=0;
}
```

Соответствующая приведенному выше фрагменту запись в ПОЛИЗе имеет вид:

```
i 0 := i 10 < 36 JZ i 5 <  j  0  >  AND 24 JZ a  i  [  j  :=
0 1 2  3 4  5 6  7  8 9 10 11 12 13 14  15 16 17 18 19 20 21

29 JMP a  i  [  0  := i  i  1  +  := 3  JMP
22 23  24 25 26 27 28 29 30 31 32 33 34 35  36
```

В заключение отметим, что в постфиксной записи недопустимо использование одних и тех же обозначений операций для унарных и бинарных операций, так как из контекста

записи невозможно определить унарность или бинарность операции. По этой причине для унарных операций следует вводить новые обозначения, либо обеспечивать эмуляцию таких операций через соответствующие бинарные операции (например, унарную операция «-a» трактовать, как бинарную «0-a»).

## *8.3 Задания*

1. Переведите следующие выражения в ПОЛИЗ вручную:

A) a+b*c

B) a*(b+c)

C) (b+c)*(d-e)

D) b*(c-d/e)

E) b+10*c-5/(d-3*e)

F) (5*b+7/c)*(9*d-e)

2. Предложите запись в ПОЛИЗе для следующих фрагментов программ:

A) a:=b*c;

B) a[i]:=b[2*i]+c[2*i+1];

C) a[i,j]:=i+j;

D) if (a>b) then c:=a else c:=b;

E) if (a*b <> 0) then begin c:=a*b; a:=0; b:=0; end;

F) while (i<100) do begin s:=s+a[i];i:=i+2; end;

3. Используя метод Замельсона – Бауэра переведите в ПОЛИЗ выражения из упражнения 1 (C-F). Задайте значения переменных a,b,c,d и рассчитайте значения выражений в ПОЛИЗе.

4. Переведите в ПОЛИЗ, используя метод Замельсона – Бауэра фрагменты программ из упражнения 3. Выполните интерпретацию полученной записи.

# 9. Трехадресный код. Тетрады и триады.

Для формирования внутреннего представления часто используется так называемый *трехадресный код (тетрады)*. В нем каждая инструкция описывается как четверка
(<оператор>,<операнд1>,<операнд2>,<результат>)

Операндами и результатом здесь могут являться переменные и константы, описанные в исходной программе или временные переменные, автоматически сгенерированные компилятором. Сам оператор во временной форме представления должен входить в набор команд, поддерживаемый выбранной моделью ЭВМ. Главное здесь, что каждая команда выполняет ровно одну базовую операцию над аргументами, находящимися в регистрах (памяти) и помещает результат также в регистр (память).

С использованием тетрад, например, следующий фрагмент кода на C++, вычисляющий сумму элементов массива:

```
s = 0;
for (i = 0; i < 10; ++i) {
    s += ar[i];
}
```

может быть представлен следующим образом:

```
      (:=, 0, , s)       // s = 0
      (:=, 0, , i)       // i = 0
L1:   (<, i, 10, t0)     // t0 = i<10
      (JZ, t0, L2,)      // if (t0 == 0) goto L2
      (+, ar, i, t1)     // t1 = ar+i
      (**, t1, , t2)     // t2=*t1, разыменование
      (+, s, t2, s)      // s=s+t2
      (+, i, 1, i)       // i=i+1
      (JMP, L1, ,)       // goto L1
L2:
```

Основным недостатком тетрад является большое количество временных переменных, порождаемых при генерации внутренней формы представления программы. Частично разрешить эту проблему помогают так называемые *триады*, имеющие следующий вид:
(<оператор>,<операнд1>,<операнд2>)

В триадах отсутствует поле результата, и если какая-либо операция должна будет воспользоваться результатом сгенерированной ранее тетрады, в качестве аргумента будет

использоваться ссылка на сгенерированную ранее триаду. Приведем пример для выражения a*b+c*d:

```
1:    (*, a, b)
2:    (*, c, d)
3:    (+,(1),(2))
```

Таким образом, тетрады и триады представляют собой довольно близкую по своей сути форму к объектному коду программы. Достоинствами тетрад и триад является относительная простота оптимизации внутренней формы представления программы, так как их можно переставлять и удалять.

## *9.1 Задания*

1. Переведите следующие выражения в тетрады:

A) b*(c-d/e)

B) (b+c)*(d-e)

C) (b+c)*(d-e)

D) b*x-v/(c-d/e)

2. Переведите следующие фрагменты программ в тетрады:

A) a = b * 4+18;

B) a[i] := 36 + f;

C) a[i,j]:=i+j;

D) if (A < N) C=B;

E) if (x==3)  a[5]=b[i]; else c[3]=0;

F) for (i=10; i<100; i++ ) s=s+a[i];

G)
```
double m = a[0];
   for (; cnt; cnt--)
      if ( a[cnt-1] > m ) m = a[cnt-1];
```

H)
```
x = -1
for (cnt--; cnt != -1; cnt--)
      if ( a[cnt] == val )  x = cnt;
```

## Литература

1. Конструирование компиляторов для цифровых вычислительных машин / Грис Д. – М.: Мир, 1975.

2. Компиляторы: принципы, технология / А. Ахо, Р. Сети, Дж. Ульман. - М.: Вильямс, 2003. - 767 с.

3. Основные структуры данных и алгоритмы компиляции / М. А. Шамашов. - Самара : СМС, 1999. - 115 с.

4. Теория и технология программирования. Основы построения трансляторов / Ю. Г. Карпов. - СПб. : БХВ-Петербург, 2005. - 270 с.

5. Теория конечных автоматов и формальных языков / Е. И. Чигарина, М. А. Шамашов. Федер. агентство по образованию, Самар. гос. аэрокосм. ун-т им. С. П. Королева. - Самара : Изд-во СГАУ, 2007. - 95 с.

6. Синтаксически ориентированный транслятор / Ингерман П. – М.: Мир, 1969.