А.С. СЕМЕНОВ

ПОРОЖДАЮЩИЕ И РАСПОЗНАЮЩИЕ СИСТЕМЫ ФОРМАЛЬНЫХ ЯЗЫКОВ

С ПРИМЕРАМИ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C#

Москва 2021

СОДЕРЖАНИЕ

Введение . . . . . .
Глава 1. Классификация грамматик и автоматов . . . .
1.1. Способы описания синтаксиса языков . . . . . .
1.2. Классификация Хомского. . . . . .
Глава 2. Автоматные грамматики и конечные автоматы . . . . . .
2.1. Определение автоматных грамматик и конечных автоматов . . .
2.2. Свойства регулярных языков: лемма о накачке . . . .
2.3. Способы задания конечных автоматов . . . .
2.4.Эпсилон-переход: реализация по заданному регулярному выражению составного автомата, построение ДКА из НКА . . . .
2.5.Пример проектирования порождающей и распознающей систем
Глава 3. Контекстно-свободные грамматики и МП-автоматы . . . .
3.1. Определение КС-грамматик и МП-автоматов . . . . . .
3.2. Преобразование КС-грамматик . . . . . .
3.3. Пример преобразования КС-грамматик к приведенной форм . .
3.4.Определение МП и расширенного МП-автоматов. Поттроение. .
3.5. Способы реализации синтаксических анализаторов . . . . . .
3.6. LL(k)-грамматики и пример построения LL(k)-анализатора . . . .
3.7. LR(k)-грамматики и пример построения LR(k)-анализатора . . .
3.8. Грамматики предшествования . . . . . .
3.9. Пример применения алгоритм “перенос-свертка” . . . . . .
Глава 4. Грамматики общего вида и машины Тьюринга . . . . . .
4.1. Грамматики общего вида и машина Тьюринга . . . . . .
4.2. Контекстно-зависимые грамматики и ленточные автоматы . .
4.3. Соотношение между грамматиками и языками . . . . . .
4.4. Способы реализации . . . . . .
Глава 5. Формальные методы описания перевода . . . . . .
5.1. Перевод и семантика . . . . . .
5.2. СУ-схемы . . . . . .
5.3. Транслирующие грамматики . . . . . .
5.4. Аттрибутные транслирующие грамматики . . . . . .
5.5. Методика разработки описания перевода. . . . . .
5.6. Пример разработки АТ грамматики . . . . . .
Глава 6. Разрработка и реализация синтаксически управляемого перевода . . . . . .
6.1. L-аттрибутивные и S-аттрибутивные транслирующие грамматика
6.2. Форма простого присваивания . . . . . .
6.3. Аттрибутивный перевод для LL(1) грамматик. . . . . .
6.4. S-аттрибутивный ДМП-процессор . . . . . .
Приложение А. Порядок выполнения задач . . . . . .
Приложение В. Задания для усвоения метериала . . . . . .
Заключение . . . . . .
Библиографический список . . . . . .

Введение

Пособие направлено на освоение и применение теории компиляции. Приводятся примеры построения синтаксических анализаторов для регулярных и контекстно-свободных грамматик, дается реализация анализаторов на объектно-ориентированном языке программирования C# [1].

Отличительной особенностью пособия является практическое содержание, включая проектирование грамматик и реализацию распознавателей. Изложенный теоретический материал используется для проведения практических работ.

Пособие состоит из трех глав и приложения, в котором приведены основные шаги выполнения практических работ и фрагменты программных текстов. Теоретический материал и практические задания организованы по темам, в соответствии с иерархией Хомского.

В первой главе даются основные определения грамматик и языков, приводятся примеры построения синтаксических анализаторов и приемы программирования.

Основной задачей практических заданий является изучение методов проектирования грамматических правил [3-5], синтаксических анализаторов и их реализации на объектно–ориентированном языке C# [2]. В процессе выполнения практических заданий развивается умение разрабатывать грамматические правила, проводить сравнительный анализ грамматик, определять их вид в соответствии с иерархией Хомского, а также применять объектно-ориентированный подход и реализовывать грамматические правила и синтаксические анализаторы на языке программирования C#.

Во второй главе рассматривается класс клеточных автоматов, которые широко рименяются в различных приложениях искусственного интеллекта и при решении дифференциальных уровнений.

В третьей главе рассмотрен класс систем, позволяющий строить различные самоподобные множества по цепочкам порожденным КС-грамматикамим. например, дерево Фибоначчи. Система определяет способ генерации совокупностей бесконечных цепочек по шаблону. Все нетерминалы замещаются параллельно. Шаблон содержит терминальные символы и определяет структуру порождаемых слов.

Приводится пример фрактального анализа дерева Фибоначчи, основывающийся на определении самоподобных структур объектов.

Глава 1. Классификация грамматик и автоматов

В главе рассматриваются два способа реализации грамматик: порождающими и распознающими системами см. рис. 1.1.

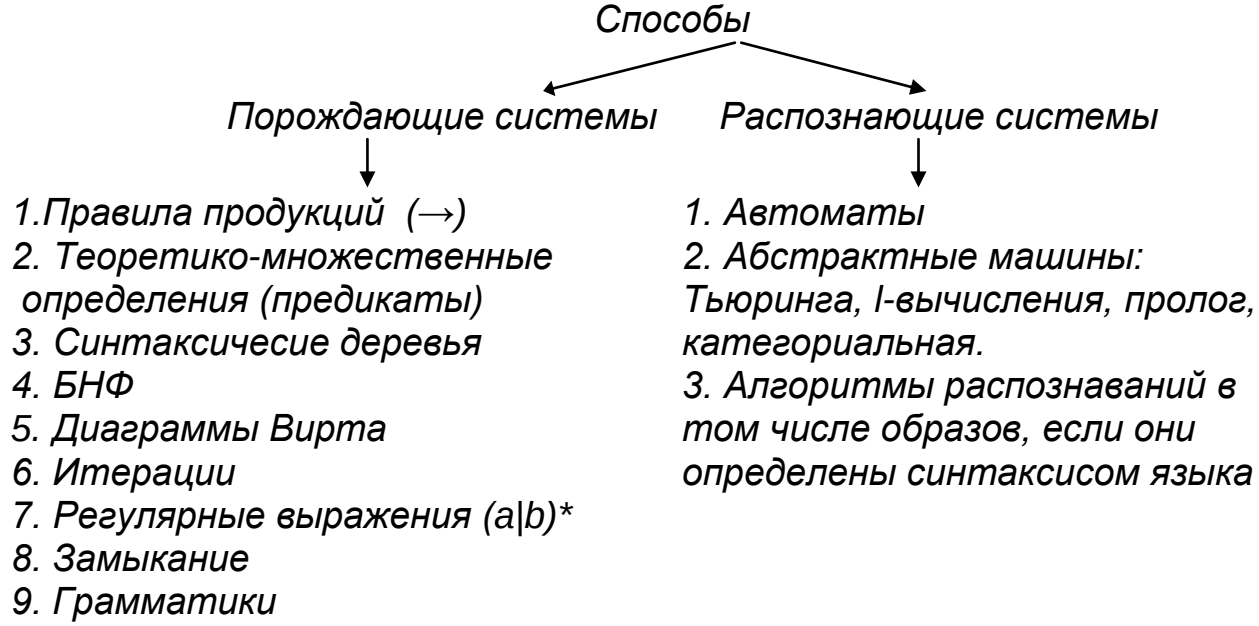


Рис. 1.1. Способы описания синтаксиса языков

Дается классификация Хомского, вводятся определения грамматик и эквивалентных им автоматов. Приводятся соотношения между типами грамматик и языков

1.1.Способы описания синтаксиса языков

Язык, будь то естественный (естественная речь) или язык программирования, задается множеством правил, образующих грамматику языка. Синтаксис языка отделяется от семантики. Появление формальных грамматик было обусловлено стремлением формализовать естественные языки и языки программирования.

Фигурные скобки используются для обозначения множеств, например {1,2,3} обозначает множество, содержащее целые числа 1, 2 и 3.

Объединение $\cup$ и пересечение $\cap$ множеств определяются как обычно:

$\{1,2,3\} \cup \{3,4,5\} = \{1,2,3,4,5\}$

$\{1,2,3\} \cap \{3,4,5\} = \{3\}$

Множество A включает $\supseteq$ множество B, если каждый элемент *B* является элементом A, например

$\{3,4,5\} \supseteq \{3\}$

Если *A* есть множество и мы определяем *B* как

$B = \{x \mid x \in N$, x есть четное число$\}$

то *B* будет множеством {2,4,8} задается с помощью предиката, в данном случае это “$x \subseteq N$, x есть четное число”.

Правила описывают те последовательности слов, которые являются корректными, т.е. допустимыми предложениями языка. Грамматика позволяет осуществить грамматический анализ предложения, а значит, и явно описать его структуру.

Одно из назначений грамматики – это определить бесконечное число предложений языка с помощью конечного числа правил.

Грамматика языка может быть реализована двумя способами: в виде порождающей системы и в виде устройств, называемых распознавателями.

1. Порождающие системы. Для описания синтаксиса языков программирования наибольшее распространение получили следующие порождающие системы: формальные грамматики, форма Бекуса-Наура и ее модификации, диаграммы Вирта [3].

В порождающих системах правила грамматики задаются продукциями, состоящими из двух частей: левой и правой. В левой части, записывается определение, в правой части варианты, из которых может состоять определение. Например, предложение “он смотрит кино” может быть определено синтаксическими правилами:

предложение состоит из *подлежащее, группа сказуемого*
подлежащее - он
группа сказуемого - *сказуемое, дополнение*
сказуемое - смотрит
дополнение – кино

Синтаксис и семантика

a b c
“он смотрит кино”

В рассматриваемых правилах порядок элементов фиксирован. Для определения грамматического формализма, эквивалентного синтаксическому, введем правило замены, которое обозначается символом “→”. Тогда порождающая грамматика, для рассматриваемого предложения, примет вид:

Предложение → подлежащее, группа сказуемого
подлежащее → он
группа сказуемого → сказуемое, дополнение
сказуемое → смотрит
дополнение → кино

Выполнив правило замены “→”, получим предложение “он смотрит кино”. Предложения языка, порождаются из исходного символа применением продукций. Шаги замены можно интерпретировать как шаги построения дерева вывода см. рис. 1.2.

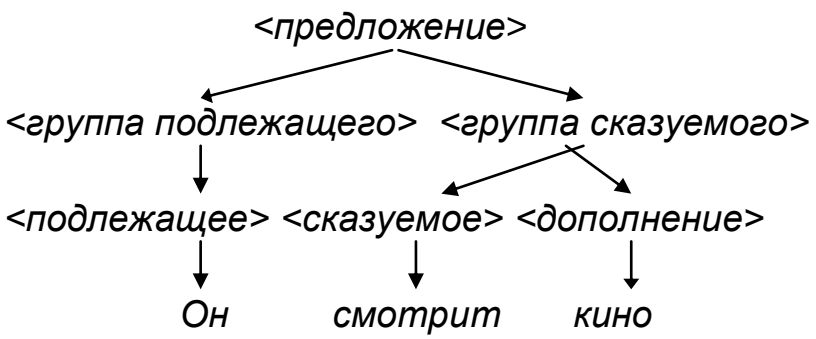


Рис. 1.2. Древовидная структура предложения

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

Обозначение правил в форме записи $\xi ::= \eta$ относится к нотации БНФ. В этой нотации используются обозначения:

::= - это есть, | - или, < > - угловые скобки, в которых записываются металингвистическая переменная, т.е. определяемое понятие;

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

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

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

БНФ для рассматриваемого примера.

<предложение>::=<подлежащее> <группа сказуемого>

<подлежащее>*::=он | она*

<группа сказуемого>::=<сказуемое><дополнение>

<сказуемое>::= *смотрит | обожает*

<дополнение>::= *кино*

Используя множество правил, *выведем* или *породим* предложение по следующей схеме. Начнем с начального символа грамматики - <предложение>, найдем правило, в котором <предложение> слева от ::= , и подставим вместо <предложение> цепочку, которая расположена справа от ::=, т.е.

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

Символ "$\Rightarrow$" означает, что один символ слева от $\Rightarrow$ в соответствии с правилом грамматики заменяется цепочкой, находящейся справа от $\Rightarrow$. Обозначение $\Rightarrow$ называют также непосредственное следование.

Заменим синтаксическое понятие на одну из цепочек, из которых оно может состоять. Повторим процесс. Возьмем один из метасимволов в цепочке <подлежащее> <группа сказуемого>, например <подлежащее>; найдем правило, где <подлежащее> находится слева от ::=, и заменим <подлежащее> в исходной цепочке на соответствующую цепочку, которая находится справа от ::=. Получим вывод:

<подлежащее> < группа сказуемого > $\Rightarrow$ *он* < группа сказуемого >

Полный вывод одного предложения будет таким:

<предложение>$\Rightarrow$<подлежащее><группа сказуемого> $\Rightarrow$ *он* <группа сказуемого> $\Rightarrow$ *он* <сказуемое> <дополнение> $\Rightarrow$ *он смотрит* <дополнение> $\Rightarrow$ *он смотрит кино*

Этот вывод предложения запишем сокращенно, используя символ $\Rightarrow^+$, который означает, что цепочка выводима *нетривиальным способом* :

<предложение> $\Rightarrow^+$ *он смотрит кино*

Последовательность цепочек $\alpha_1, \alpha_2, \ldots, \alpha_n$ в этом случае называется выводом цепочки α_n из α_1**:** $\alpha_1 \Rightarrow^+ \alpha_n$.

Две последовательности связаны отношением $\Rightarrow^*$, когда вторая получается из первой применением некоторой последовательности продукции $\alpha_1 \Rightarrow^* \alpha_n$ тогда,

и только тогда, когда $\alpha \Rightarrow^{+} \alpha_n$. $\alpha_1 \Rightarrow^{*} \alpha_n$ означает, что цепочка α_n выводится из α_1 за **ноль** или более шагов.

На каждом шаге можно заменить любую метапеременную. В приведенном выше выводе всегда заменялся самый левый из них. Если в процессе вывода цепочки правила грамматики применяются только к самому левому нетерминалу, говорят, что получен *левый вывод* цепочки. Аналогично определяется правый вывод.

Рассматриваемая грамматика определяет несколько предложений (цепочек) языка:

он смотрит кино *он обожает кино*
она смотрит кино *она обожает кино*

Диаграммы Вирта позволяют визуально представить правила порождения. Каждому правилу соответствует диаграмма.

Элементы ИЛИ обозначаются разветвлениями и вершинами, элементы И последовательными вершинами графа, метасимволы {...} циклом. Вершины могут быть двух типов: терминальные – кружки, и нетерминальные – прямоугольники. Каждому правилу соответствует диаграмма см. рис. 1.3.

Правила- И, цикл, ИЛИ: T → { F } | (**E**)

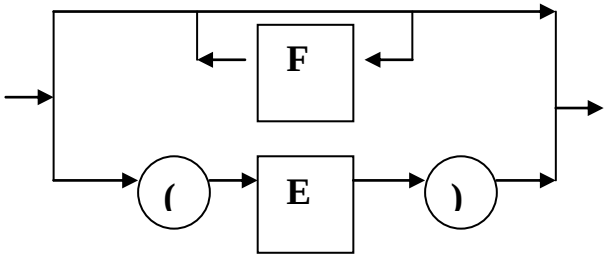


Рис. 1.3. Диаграммы Вирта

Итерационная форма описания. Вводится операция итерации – обозначается парой круглых скобок со звездочкой.

Итерация вида $(a)^{*}$ определяется как множество, включающее цепочки всевозможной длины, построенные с использованием символа **a**.

$(a)^{*}$ = {a, aa, aaa, aaaa, …}

Определим понятия, используемые при определении грамматик формальных языков и распознавателей.

Алфавит (или словарный состав) – конечное множество символов. Например, *алфавит* V_1 = {a,b} – содержит два символа, *алфавит* V_2 = {01,11,10,00} – содержит четыре символа.

Цепочкой символов α *в алфавите* V называется любая конечная последовательность символов этого алфавита.

Пустой цепочкой символов ε называется цепочка, не содержащая ни одного символа.

Длина цепочки α – число составляющих ее символов, обозначается | α |. Например, если цепочка символов α = defgabck, то длина α равна 8, | α | = 8. Длина ε равна 0, | ε | = 0.

Замыка́ние Кли́ни (или звезда́ Кли́ни) в математической логике и информатике – унарная операция над множеством строк либо символов. Замыкание Клини множества V обозначается V*. Широко применяется в

регулярных выражениях, на примере которых было введено Стивеном Клини для описания некоторых автоматов.

Множество всех строк (включая пустую), которые могут быть построены из **символов** алфавита V, называется **замыканием** V, и обозначается V*.

Пусть V – множество строк, тогда V* – минимальное надмножество множества V, которое содержит ε (пустую строку) и замкнуто относительно конкатенации.

Надмножество (superset) – множество, подмножеством которого является данное. Иначе супермножество, объемлющее множество (множество множеств).

V^* (замыкание V*)* – обозначается множество всех цепочек составленных из символов алфавита V, включая пустую цепочку ε. Это также множество всех строк, полученных конкатенацией ε или более строк из V. Например, если V={0,1}, то $V^* = \{\varepsilon, 0, 1, 00, 11, 01, 10, 000, 001, 011, ...\}$.

V^+ – обозначается множество всех цепочек составленных из алфавита V, исключая пустую цепочку ε. Следовательно, $V^* = V^+ \cup \{\varepsilon\}$.

Декартовым произведением A×B множеств A и B называется множество $\{(a,b) \mid a \in A, b \in B\}$.

∅ – обозначается пустое множество.

2. Распознающие системы (автоматы). Грамматика может быть реализована в виде некоторого алгоритма, производящего действия и отвечающего на вопрос, принадлежит ли данная фраза языку.

Функцией перехода δ называется отношение, управляющее действиями распознавателя. Функция перехода δ определяет для каждой пары (входной символ, состояние) множество состояний, в которых он будет находится на следующем шаге.

Например, принадлежит ли предложение “он смотрит кино” языку заданному рассмотренными грамматическими правилами? Для этого необходимо построить распознаватель.

1.2. Классификация Хомского

Синтаксис языка L можно специфицировать, пользуясь системой изображения множеств, например:

$L = \{0^n1^n \mid n \geq 0\}$

Язык L включает строки, состоящие из нуля или нулей и того же числа последующих единиц. Пустая строка включается в язык.

Такое задание синтаксиса намного проще синтаксисов большинства языков специфицируемых с помощью *грамматики.*

Определение 1. Грамматика $G = (T, V, P, S_0)$,

где T - конечное множество терминальных символов (терминалов) алфавита;

V – конечное множество нетерминальных символов алфавита, не пересекающихся с T, $T \cap V = \varnothing$,

S_0 - начальный символ (или аксиома), $S_0 \in V$;

P - конечное множество правил порождения (продукций), $P = (T \cup V)^+ \times (T \cup V)^*$. Элемент (α, β) множества P называется *правилом порождения* и записывается в виде $\alpha \to \beta$.

Правила порождения с одинаковыми левыми частями $\alpha \to \beta_1$, $\alpha \to \beta_2$,..., $\alpha \to \beta_n$, записывают сокращенно $\alpha \to \beta_1 \mid \beta_2 \mid ... \mid \beta_n$, где β_i , i= 1, 2,..., n и называют *альтернативой* правил порождения из цепочки α.

Выводом (обозначим $\Rightarrow$) в грамматике *G* называют произвольную, конечную или бесконечную, последовательность цепочек. Цепочка $\beta \subset (T \cup V)^*$ *выводима* из цепочки $\alpha \subset (T \cup V)^+$ в грамматике G, ($\alpha \Rightarrow \beta$), если существуют цепочки $\gamma_0, \gamma_1, ..., \gamma_n$ ($n \geq 0$), такие, что $\alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow ... \Rightarrow \gamma_n = \beta$. Последовательность $\gamma_0, \gamma_1, ..., \gamma_n$ называется *выводом длины n*.

Цепочка $\beta \subset (T \cup V)^*$ *непосредственно выводима* ($\alpha \Rightarrow \beta$) из цепочки $\alpha \subset (T \cup V)^+$ в грамматике G, если $\alpha = \xi_1\gamma\xi_2$, $\beta = \xi_1\delta\xi_2$, где $\xi_1, \xi_2, \delta \subset (T \cup V)^*$, $\gamma \subset (T \cup V)^+$ и правило продукции $\gamma \to \delta$ содержится в P.

Для обозначения терминалов грамматики мы будем употреблять прописные буквы (или строки прописных букв), а для обозначения не терминалов — заглавные буквы (или строки заглавных букв).

Примеры грамматик.

Пример 1. Для примера “он смотрит кино” имеем грамматику

G = ({“он”, “кино”, “смотрит”},{*Предложение, подлежащее, группа сказуемого, сказуемое, дополнение*}, P, S_0), где S_0 = *Предложение*, P состоит из правил $P = \{p_1, p_2, p_3, p_4, p_5\}$;

p_1: *Предложение* → *подлежащее, группа сказуемого*

p_2: *подлежащее* → он

p_3: *группа сказуемого* → *сказуемое, дополнение*

p_4: *сказуемое* → смотрит

p_5: *дополнение* → кино

Пример 2. G = ({c,d}, {B,S_0}, P, S_0), где P состоит из правил

$S_0 \to cBd$

$cB \to ccBd$

$B \to \varepsilon$

Цепочка ccBdd непосредственно выводима из cBd в грамматике G.

Цепочка cccBddd в грамматике G выводима, т.к. существует вывод $S_0 \Rightarrow cBd \Rightarrow ccBdd \Rightarrow cccBddd \Rightarrow ccdddd$. Длина вывода равна 4.

Определение 2. Языком, порождаемым грамматикой G = (T, V, P, S_0), называется множество $L(G) = \{\alpha \subset T^* \mid S_0 \Rightarrow^* \alpha\}$.

Другими словами, L(G) - это все цепочки в алфавите T, которые выводимы из S_0 с помощью P.

Цепочка $\alpha \subset (T \cup V)^*$, для которой $S_0 \Rightarrow^* \alpha$ (то есть цепока, которая может быть выведена из начального символа), называется *сентенциальной формой* в грамматике G. Язык, порождаемый грамматикой, можно определить как множество терминальных сентенциальных форм.

Например, пусть задана грамматика $G_1 = (\{0,1\}, \{A, S_0\}, P_1, S_0)$, состоящая из правил P_1: $S_0 \to 0A1$

$0A \to 00A1$

$A \to \varepsilon$

Тогда язык, порождаемый грамматикой G_1, это язык $L(G_1) = \{0^n1^n \mid n>0\}$.

Грамматики G_1 и G_2 называются *эквивалентными*, если $L(G_1) = L(G_2)$. Например, пусть задана грамматика $G_2 = (\{0,1\}, \{S\}, P_2, S)$, где

P_2: $S \to 0S1 \mid 01$, тогда

грамматики G_1 и G_2 эквивалентны, т.к. обе порождают язык $L(G_1) = L(G_2) = \{0^n1^n \mid n>0\}$.

Грамматики G_2 и G_3 называются *почти эквивалентными* $L(G_2) = L(G_3)$, если $L(G_2) = L(G_3) \cup \{\varepsilon\}$. Другими словами, грамматики почти эквивалентны, если языки, ими порождаемые, отличаются не более чем на ε.

Определим грамматику $G_3 = (\{0,1\}, \{S\}, P_3, S)$ с правилами

P_3: $S \to 0S1 \mid \varepsilon$

Таб. 1.1. Классификация Хомского

Тип	Ограничения на правила P	Пример правил грамматики $G = (T, V, P, S_0)$	Пример языка	Автомат или абстрактная машина
0	Общего вида, не накладывается никаких ограничений	$S_0 \to CD$, $C \to 0CA$, $C \to 1CB$, $AD \to 0D$, $BD \to 1D$, $A0 \to 0A$, $A1 \to 1A$, $B0 \to 0B$, $B1 \to 1B$, $C \to \varepsilon$, $D \to \varepsilon$	$\{\omega\omega \mid \omega \in \{0,1\}^*$, язык состоит из цепочек четной длинны, из 0 и 1$\}$	Машина Тьюринга
1	Контекстно-зависимая (КЗ) $\alpha \to \beta$, $\mid \alpha \mid \le \mid \beta \mid$	$S_0 \to 0B0$ $B \to 1 \mid 0BC$ $C0 \to 100$ $C1 \to 1C$	$\{0^n1^n0^n \mid n \ge 1\}$	Машина Тьюринга с конечной лентой
2	Контекстно-свободная (КС) $A \to \alpha$	$S_0 \to AS_0B \mid AB$ $A \to 0$ $B \to 1$ LL(k) грамматика LR(k) грамматика	$\{0^n1^n \mid n \ge 1\}$	с магазинной памятью (МП-автомат: LL-разбор, РМП-автомат: LR-разбор) Распознаватели: LL(1) LR(1) подкласс LR(1) грамматики предшествования
3	Регулярная $A \to aB$ $B \to b$	$S_0 \to 0 \mid 0A$ $A \to 0B$ $B \to 1 \mid 1A$	$\{0(01)^n \mid n \ge 0\}$	Конечный автомат (КА)

Тогда грамматики G_2 и G_3 *почти эквивалентны*, так как $L(G_2)=\{0^n1^n \mid n>0\}$, а $L(G_3)=\{0^n1^n \mid n\geq 0\}$, т.е. $L(G_3)$ состоит из всех цепочек языка $L(G_2)$ и пустой цепочки, которая в $L(G_2)$ не входит.

Классификация Хомского в таб.1.1. дает представление о 4 типах грамматик, располагающихся по иерархии от 0 до 3 в порядке убывания их общности, и о эквивалентных формальных автоматах.

Грамматики классифицируются по виду правил вывода P. В правилах вывода P большими латинскими буквами обозначают нетерминальные символы, маленькими латинскими буквами - терминальные.

Автомат эквивалентен грамматике, если он воспринимает весь порожденный грамматикой язык и только этот язык.

Классификация Хомского дает представление об общности грамматик, и об эквивалентных им формальных автоматах.

Глава 2. Автоматные грамматики и конечные автоматы

2.1. Определение автоматных грамматик и конечных автоматов

Определение 3. Грамматика типа 3 или автоматная (регулярная) А-грамматика – это грамматика $G = (T, V, P, S_0)$, у которой правила порождения P имеют вид по классификации Хомского $A \rightarrow bB$ (праволинейное правило) или $B \rightarrow a$ (заключительное правило), где $A, B \in V$, $a,b \in T$. Каждое правило такой грамматики содержит единственный нетерминал в левой части, всегда один терминал в правой части, за которым может следовать один нетерминал. Такую грамматику также называют *праволинейной*.

Грамматика $G = (T, V, P, S_0)$ называется *леволинейной*, если каждое правило из P имеет вид по классификации Хомского $A \rightarrow Bb$ либо $B \rightarrow a$, где $A, B \in V$, $a,b \in T$.

Таким образом, грамматику типа 3 можно определить как праволинейную либо как леволинейную.

Выбор определения не влияет на множество языков, порождаемых грамматиками этого класса, поскольку доказано, что множество языков, порождаемых праволинейными грамматиками, совпадает с множеством языков, порождаемых леволинейными грамматиками.

Автоматной грамматике эквивалентен простейший распознаватель – недетерминированный конечный автомат. Автомат представляет собой устройство, у которого отсутствует вспомогательная память.

Определение 4. Конечный автомат (КА) – это пятерка объектов

$KA = (Q, \Sigma, \delta, q_0, F)$, где

Q - конечное множество состояний;

Σ - конечный алфавит входных символов;

δ - функция переходов, задаваемая отображением

$$\delta: Q \times \Sigma \rightarrow Q,$$

где Q - конечное множество подмножеств множества Q;

q_0 - начальное состояние автомата, $q_0 \in Q$;

$F \subseteq Q$ - множество заключительных состояний.

В каждый момент времени КА находится в некотором состоянии $q \in Q$ и читает поэлементно последовательность символов $a_k \in \Sigma$, записанную на конечной ленте. При этом либо читающая головка машины двигается в одном направлении (слева направо), либо лента перемещается (справа налево) рис.2.1. при неподвижной читающей головке.

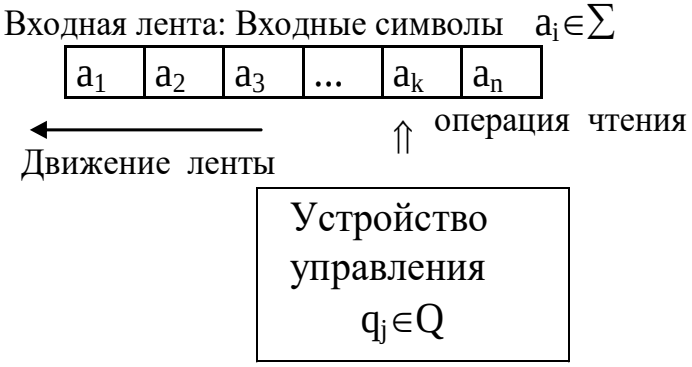


Рис. 2.1. Схема конечного автомата

Входная лента: Входные символы $a_i \in \sum$

Если автомат в состоянии q_i читает символ a_k и определена функция перехода $\delta(q_i,a_k) = q_j$, то автомат воспринимает символ a_k и переходит в состояние q_j для обработки следующего символа.

Определение 5. Конфигурация КА – это пара множества $(q, \omega) \in Q \times \sum^*$, где $q \in Q$, $\omega \in \sum^*$. Конфигурация (q_0, ω) называется *начальной*, а (q, ε), где $q \in F$, - заключительной.

Определим бинарное отношение $\vdash$ на конфигурациях, соответствующее одному такту работы КА. Если $q' \in \delta(q,a)$, то

$(q, a\omega) \vdash (q', \omega)$ для всех $\omega \in \sum^*$.

Пусть {C} – множество конфигураций.

1. $C \vdash^0 C'$ означает, что $C = C'$
2. $C_0 \vdash^k C_k$, если существует последовательность конфигураций $C_1, C_2, \ldots, C_{k-1}$, для $k \geq 1$, в которых $C_i \vdash C_{i+1}$ для $0 \leq i < k$.
3. $C \vdash^+ C'$ означает, что $C \vdash^k C_k$ для некоторого $k \geq 1$, а $C \vdash C'$ означает, что $C \vdash^k C'$ для $k \geq 1$.

Конечеый автомат КА = $(Q, \sum, \delta, q_0, F)$ распознает входную цепочку $\omega \in \sum^*$, если $(q_0, \omega) \vdash^k (q, \varepsilon)$ для $q \in F$.

Языком L(КА), распознаваемым КА называется множество входных цепочек,

$L(КА) = \{\omega \in \sum^* | (q_0,\omega) \vdash^k (q, \varepsilon)$, где $q \in F\}$

КА называется *недетерминированным*, если для каждой конфигурации существует конечное множество всевозможных следующих шагов, любой из которых КА может сделать, исходя из этой конфигурации.

КА называется *детерминированным*, если для каждой конфигурации существует не более одного следующего шага.

Для любого конечного недетерминированного автомата можно построить ему эквивалентный детерминированный автомат.

Класс языков, распознаваемый конечными автоматными, совпадает с классом языков, порождаемых автоматными грамматиками и наоборот.

Утверждение 1. Пусть задана автоматная грамматика $G = (T, V, P, S_0)$, тогда существует такой (недетерминированный) конечный автомат КА = $(Q, \sum, \delta, q_0, F)$, что $L(КА) = L(G)$. КА строится следующим образом:

1. Входные алфавиты конечного автомата $\sum$ и автоматной грамматики T совпадают, $\sum = T$.
2. $Q = V \cup \{q_f\}$, где q_f – заключительное состояние конечного автомата.

3. $q_0 = S_0$.

4. Если $S_0 \to \varepsilon \in P$, то $F = \{ S_0, q_f \}$, в противном случае $F = \{ q_f \}$.

5. Функция переходов КА определяется следующим образом:

1. $q_f \in \delta(B,a)$, если $B \to a \in P$, $B \in V$, $a \in \sum$;
2. если $B \to aC \in P$, то $C \in \delta(B,a)$;
3. $\delta(q_f , a) = \varnothing$ для всех $a \in \sum$.

Пример. Пусть задан регулярный язык $L=\{0(10)^n \mid n \geq 0\}$. Построить автоматную грамматику $G = (T,V,P,S_0)$ для заданного языка L и привести пример вывода строки. Используя грамматику G, построить

КА= $(\sum,Q,\delta, q_0, F)$ и привести пример конфигурации КА.

1. Построение грамматики. $L=\{0(10)^n \mid n \geq 0\}$, то 0, 010, 01010 и т.д. - этот язык порожден регулярной грамматикой $G = (T,V, P,S_0)$, где $T=\{0, 1\}$, $V=\{S_0,A,B\}$, $P=\{S_0 \to 0, S_0 \to 0A, A \to 1B, B \to 0, B \to 0A\}$. Пример вывода цепочки

$S_0 \Rightarrow 0A \Rightarrow 01B \Rightarrow 010A \Rightarrow 0101B \Rightarrow 01010$.

2. Построение КА. Воспользуемся утверждением 1, тогда

КА= $(\{0, 1\}, \{S_0, A, B, q_f\}, \delta, S_0, q_f)$

$\delta(S_0, 0) = \{A, q_f \}$

$\delta(A, 1) = \{B\}$

$\delta(B, 0) = \{A, q_f \}$

Пример конфигурации КА: $(S_0,01010) \vdash (A,1010) \vdash (B,010) \vdash (A,10) \vdash (B,0) \vdash (q_f,\varepsilon)$

2.2. Способы задания конечных автоматов

На рис. 2.2. приведена диаграмма переходов КА для рассматриваемого примера.

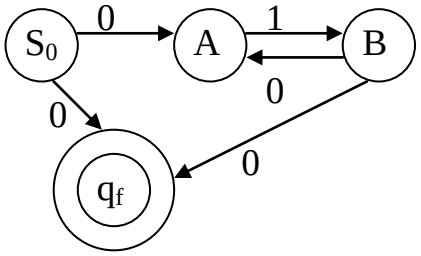


Рис. 2.2. Диаграмма переходов КА

Конечный автомат можно задать в виде таблицы переходов и диаграммы (графа) переходов.

В таблице переходов двум аргументам ставится в соответствие одно значение. В таб. 2.1 приведена таблица переходов КА для рассматриваемого примера.

Таб. 2.1. Таблица переходов для КА

	0	1
S_0	$\{ A, q_f\}$	
A		{B}
B	$\{A, q_f \}$	

q_f		

Диаграммой переходов КА называется неупорядоченный граф, удовлетворяющий условиям:

- каждому состоянию q соответствует некоторая вершина, отмеченная его именем;
- диаграмма переходов содержит дугу из состояния q_k в состояние q_n , отмеченную символом a, если $q_k \in \delta(q_n, a)$. Дуга может быть помечена множеством символов, переводящих автомат из состояния q_k в состояние q_n;
- вершины, соответствующие заключительным состояниям $q_f \in F$, отмечаются двойным кружком.

2.3. Свойства регулярных языков: лемма о накачке

Теорема, называемая “лемма о накачке” утверждает, что все достаточно длинные цепочки регулярного языка можно *накачать*, то есть **повторить** внутреннюю часть цепочки символов сколько угодно раз, производя новую подцепочку, также принадлежащее языку.

Все конечные языки являются автоматными, эту проверку имеет смысл делать только для бесконечных языков.

Лемма описывает существенное свойство всех регулярных языков и служит инструментом для доказательства нерегулярности некоторых языков.

Формальное утверждение. Длина накачки p принимается более длины самой длинной цепочеки языка L, такое что цепочка w из L длины по меньшей мере p может быть записана как w = xyz, где y – это подцепочка, которую можно накачать (удалить или повторить произвольное число раз, так что результат останется в L).

(1) Для **всех** регулярных языков $\forall$ L $\subseteq \Sigma^*$(regular(L) =>

(2) существует целое ($\exists$ p ≥ 1 такое что

(3) для **всех** ($\forall$ $w \in$ L (($|w| \geq p$) =>

(4) существует ($\exists$ $x,y, z \in \Sigma^*$ такое что ($w = xyz$ =>

1. ($| y | \geq 1$, цикл y должен быть накачан хотя бы длиной 1 и
2. $|xy | \leq p$, цикл должен быть в пределах первых p символов и
3. для всех $i \geq 0$, ($xy^iz \in$ L))))))), на x и z ограничений не накладывается.

$$\forall L \subseteq \Sigma^*(\text{regular}(L) \Rightarrow (\exists p \geq 1(\forall w \in L((|w| \geq p) \Rightarrow (\exists x,y,z \in \Sigma^*(w = xyz \Rightarrow (|y| \geq 1 \wedge |xy| \leq p \wedge \forall i \geq 0(xy^iz \in L)))))))$$

Для доказательства регулярности языка:

1. выделить цепочку $\boldsymbol{y}^{\mathbf{i}}$ для всех $i \geq 0$, $x\boldsymbol{y}^{\mathbf{i}}z \in L$, на x и z ограничений не накладывается (то есть, нет цепочки символов $\boldsymbol{y}^{\mathbf{i}}$, то не регулярный язык).
2. построить распознаватель для заданного языка - конечный автомат;

Свойство замкнутости регулярных языков, позволяет минимизировать построение автомата:

1. строить рапознаватели для одних языков, построенных из других с помощью операций;
2. определить, что два различных автомата определяют один язык.

Пример 2. Рассмотрим язык $L_{01} = \{0^n1^n \mid n \geq 1\}$, состоящий из всех цепочек вида 01, 0011, 000111 и так далее, содержащий один или несколько нулей, за которыми следует такое же количество единиц.

Утверждается, что язык L_{01} нерегулярен.

Если бы L_{01} был регулярным языком, то допускался некоторым ДКА, имеющим какое-то число состояний k. Пусть на вход ДКА поступает k нулей. Он находится в нкотором состоянии после чтения каждого из k+1 префиксов входной цепочки, т.е. ε, 0,00,…,0^k. Поскольку есть только k различных состояния, например, 0^i, 0^j , автомат должен находиться в одном и том же состоянии.

Прочитав i или j нулей ДКА получает на вход 1. По прочтении i единиц он должен допустить вход, если ранее получил i нулей, и отвергнуть его, получив j нулей. Но в момент поступления 1, автомат не способен вспомнить какое число нулей i, j было принято. Следовательно, он может работать неправильно.

Пример 3. Нерегулярность языка $L = \{a^n b^n \mid n \geq 0\}$ над алфавитом $\Sigma = \{a, b\}$ можно показать следующим образом.

Пусть *w, x, y, z, p*, и *i* заданы соответственно формулировке леммы выше. Пусть *w* из *L* задаётся как $w = a^p b^p$. По лемме о накачке, существует разбиение $w = xyz$, где $|xy| \leq p$, $|y| \geq 1$, такое что xy^iz принадлежит L для любого $i \geq 0$. Если допустить, что $|xy|=p$, а $|z|=p$, то xy — это первая часть *w*, состоящая из *p* последовательных экземпляров символа *a*.

Поскольку $|y| \geq 1$, она содержит по меньшей мере одну букву *a*, а xy^2z содержит больше букв *a* чем *b*. Следовательно, xy^2z не в языке *L* (заметим, что любое значение $i \neq 1$ даст противоречие). Достигнуто противоречие, поскольку в этом случае накачанное слово не принадлежит языку *L*. Предположение о регулярности *L* неверно и *L* – не регулярный язык.

Пример 1. Пусть КА (см. рис.) распознает строку **abcbcd**. Поскольку длина её превышает число состояний, существуют повторяющиеся состояния: q1 и q2.

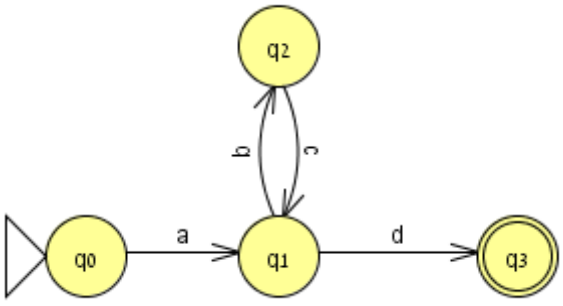


Рис. 2.3. Диаграмма переходов КА

Поскольку подстрока **bcbc** строки **abcbcd** проводит автомат по переходам из состояния q_1 обратно в q_1, эту строку можно повторять сколько угодно раз, и КА всё равно будет её принимать, например строки **abcbcbcbcd** и **ad**.

В терминах леммы о накачке, строка **abcbcd** разбивается на часть *x* = **a**, часть *y* = **bc** и часть *z* = **bcd**.

Заметим, что можно разбить её различными способами, например x = **a**, y = **bcbc**, z = **d**, и все условия будут выполнены, кроме $|xy| \le p$, *где p – длинна накачки*.

Доказательство леммы о накачке

Для каждого регулярного языка существует конечный автомат (КА), распознающий этот язык.

Если язык конечен, то результат можно получить немедленно, задав длину накачки p более длины самого длинного слова языка: в этом случае нет ни одной цепочки в языке длиннее p, и утверждение леммы выполняется безусловно.

Если регулярный язык бесконечен, то существует минимальный КА, распознающий его. Число состояний этого КА и принимается за длину накачки p. Если длина цепочки превышает p, то хотя бы одно состояние при обработке цепочки повторяется (назовём его S). Переходы из состояния S и обратно соответствуют некоторой цепочки. Эту цепочку обозначим y по условиям леммы, и поскольку автомат примет строку как без части y, так и с повторяющейся частью y, условия леммы выполнены.

Доказательство. Пусть L – регулярный язык, тогда L = L(ДКА) для некоторого ДКА. Пусть ДКА имеет n состояний. Рассмотрим произвольную цепочку w длинной не менее n. и скажем, $w = a_1a_2a_3..a_m$, где $m \ge n$ и каждый a_i есть входной символ. Для i = 0,1,2,…,n определим состояние p_i как $\delta(q_0, a_1a_2a_3..a_i)$, где δ – функция переходов автомата, q_0 – его начальное состояние.

Заметим, что $p_0 = q_0$.

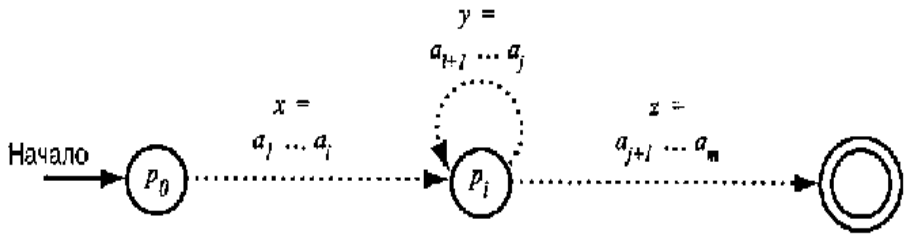
Рассмотрим n+1 состояний p_i при i = 0,1,2,…,n. Поскольку автомат имеет n различных состояний, то найдутся два разных целых числа i и j ($0 \le i < j \le n$), при которых $p_i = p_j$. Теперь разобьем цепочку w на xyz.

$x = a_1a_2...a_i$

$y = a_{i+1}a_{i+2}...a_j$

$z = a_{j+1}a_{j+2}...a_m$

Таким образом, x приводит в состояние p_i, y – из p_i обратно в p_i (так как $p_i = p_j$), а z – это остаток цепочки w. Взаимосвязи между цепочками и состояниями показаны на рис. Заметим, что цепочка x может быть пустой при i=0, а z – при j=n=m. Однако цепочка y не может быть пустой, поскольку i строго меньше j.



ис. 4.1. Каждая цепочка, длина которой больше числа состояний автома
приводит к повторению некоторого состояния

Рис. 2.4. Каждая цепочка, длина которой больше числа состояний автомата приводит к повторению некоторого состояния

На вход автомата поступает цепочка xy^kz для любого $k \geq 0$.

1.При $k = 0$ автомат переходит из начального состояния q_0 (которое есть также p_0) в p_i прочитав x. Поскольку $p_i = p_j$, то z переводит автомат из p_i в допускающее состояние (рис. 4.1.).

2.Если $k > 0$, то по x автомат переходит из q_0 в p_i, затем читая y^k, k раз циклически проходит через p_j, и, наконец, по z переходит в допускающие состояния.

3.Для любого $k \geq 0$ цепочка xy^kz также допускается автоматом, то есть принадлежит языку L.

2.4. Эпсилон-переход: реализация по заданному регулярному выражению составного автомат, построение ДКА из НКА

Автомат переходит из состояния в состояние с помощью функции перехода δ, читая при этом один символ из ввода. Есть автоматы, которые могут перейти в новое состояние без чтения символа.

Определение. Функция перехода без чтения символа называется *ε-переход (эпсилон-переход).*

Определение. *ε-замыканием* состояния s_i называется множество состояний НКА, в которые из s_i можно попасть по цепочке ε-переходов. Как минимум, в это множество входит само s_i .

Определение. *move(Q, a)* - функция переходов, возвращает состояние, в которое происходит переход из состояния $q \in Q$ при входном символе a.

ε-переход – позволяет объеденить нескольких автоматов в один. Так, например, два автомата, представленных на рис.2.5, можно соединить с помощью ε-переходов. В итоге получим автомат, представленный на рис.2.6.

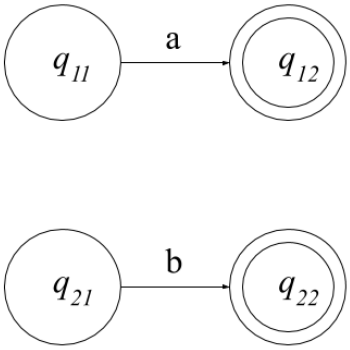


Рис.2.5

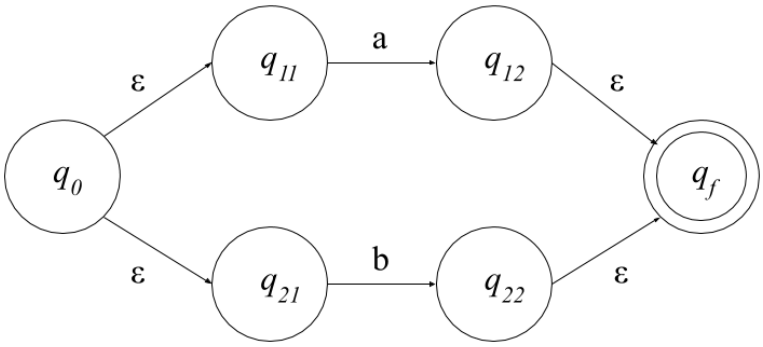


Рис.2.6

Алгоритм. Построение ДКА из НКА (алгоритм называют "построение подмножества")

В таблице переходов НКА каждая запись представляет собой множество состояний; в таблице переходов ДКА — единственное состояние. Общая идея преобразования НКА в ДКА состоит в том, что каждое состояние ДКА соответствует множеству состояний НКА. ДКА использует свои состояния для отслеживания всех возможных состояний, в которых НКА может находиться после чтения очередного входного символа. Таким образом, после чтения входного потока $a_1a_2..a_n$. ДКА находится в состоянии, которое ставляет подмножество T состояний НКА, достижимых из стартового состояния НКА по пути, помеченному как $a_1a_2..a_n$. Количество состояний ДКА может оказаться экспоненциально зависящим от количества состояний НКА, но на практике этот наихудший случай встречается крайне редко.

Вход. НКА $N = (Q, \sum, \delta, q_0, F)$.
Выход. ДКА $D = (Q', \sum, \delta', q_0', F')$, допускающий тот же язык.

Метод. Построим таблицу переходов *Dtran* для ДКА. Каждое состояние ДКА является множеством состояний НКА, и мы строим *Dtran* так, чтобы ДКА “параллельно” моделировал все возможные перемещения НКА по данной входной строке.

Для отслеживания множеств состояний НКА используем операции, приведенные на рис.2.7 (*q* представляет состояние НКА, а *Q* - множество состояний НКА).

Операция	Описание
ε-closure(q) (ε-замыкание(q))	Множество состояний НКА, достижимых из состояния *q* только по *ε*-переходам
ε-closure(Q) (ε-замыкание(Q))	Множество состояний НКА, достижимых из какого-либо состояния *q* из *Q* только по *ε*-переходам
move(Q, a)	Множество состояний НКА, в которые имеется переход из какого-либо состояния *q* из *Q* по

	входному символу *a*

Рис. 2.7. Операции над состояниями НКА

Перед тем как рассматривать первый входной символ, НКА может быть в любом из состояний множества *ε-closure(q_0),* где q_0 - стартовое состояние НКА. Предположим, что состояния множества *Q*, и только они, достижимы из q_0 после прочтения данной последовательности входных символов, и пусть *a*— следующий входной символ. Получив *a*, НКА может переместиться в любое состояние из множества *move(Q, a)*. Совершив из этих состояний все возможные *ε*-переходы, НКА после обработки *a* может быть в любом состоянии из *ε-closure(move(Q, a)).*

Множество состояний *Dstates* автомата ДКА и таблицу его переходов *Dtran* можно создать следующим образом:

1. Каждое состояние ДКА соответствует множеству состояний НКА, в которых может находиться НКА после чтения некоторой последовательности входных символов, включая все возможные *ε*-переходы до и после считанных символов.

2. Стартовое состояние ДКА- *ε-closure(q_0).*

3.Состояния и переходы добавляются в ДКА согласно следующему алгоритму:

Изначально *ε-closure(q_0)* является единственным состоянием в *Dstates* и не помечено;
while в *Dstates* имеется не помеченное состояние *Q* **do**
    пометить *Q*;
    for каждый входной символ *a* **do**
        U := ε-closure(move(Q, a));
        if $U \notin Dstates$ **then**
            Добавить *U* как не помеченное состояние в *Dstates*;
        Dtran[Q, a] := U
    end
end

Рис. 2.8. Построение подмножества

4. Состояние ДКА является допускающим, если оно представляет собой множество состояний НКА, содержащих как минимум одно допускающее состояние НКА.

Вычисление *ε-closure(Q)* является типичным процессом поиска графа для узлов, достижимых из данного множества узлов. В этом случае состояния *Q* представляют данное множество узлов, а граф состоит только из дуг НКА, помеченных *ε*. Простой алгоритм вычисления *ε-closure(Q)* использует стек для хранения состояний, дуги которых не были проверены на наличие *ε*-переходов. Такая процедура показана на рис. 2.9.

Внести все состояния множества *Q* в стек *stack* ;

инициализировать *ε-closure(Q)* множеством *Q*;
while *stack* не пуст **do**
    снять со стека верхний элемент *t*
    for каждое состояние *u* с дугой
        от *t* к *u*, помеченной *ε* **do**
    if *u* ∉ *ε -closure(Q)* **then**
        добавить *u* к *ε-closure(Q)* ;
        поместить *u* в *stack*
    end
end

Рис. 2.9. Вычисление ε-замыкания

Пример 1. На рис. 2.10 показан НКА *N*, допускающий язык $(a \mid b)^* abb$.

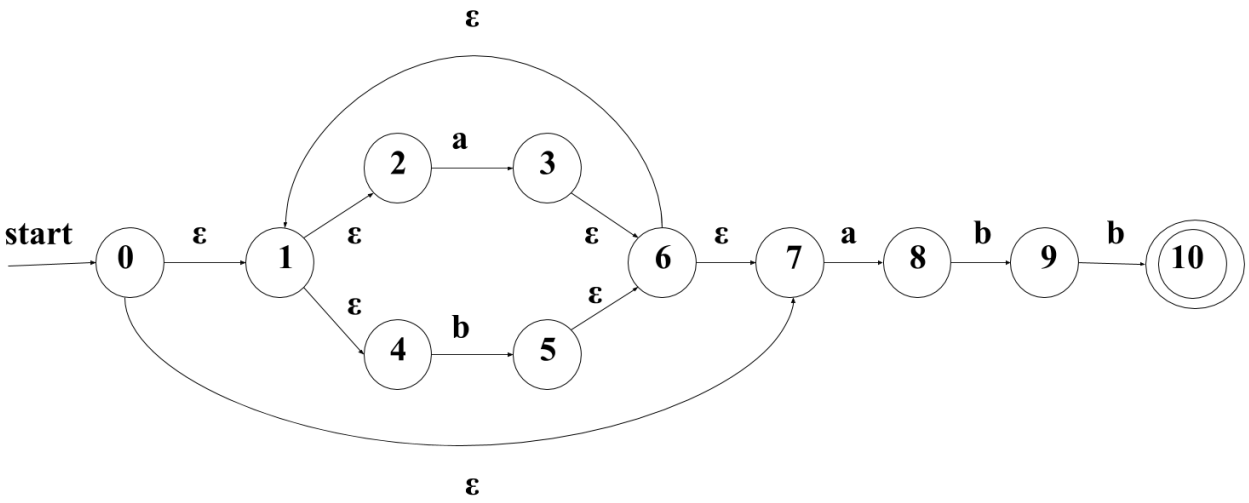


Рис. 2.10. НКА N для $(a \mid b)^* abb$

Применим к *N* алгоритм.

Алфавит входных символов представляет собой *{a, b}*.

Стартовое состояние эквивалентного ДКА представляет собой *ε-closure(0)*, т.е. *A={0, 1, 2, 4, 7}*, поскольку именно эти состояния достижимы из состояния *0* путями, в которых каждая дуга помечена *ε*. Заметим, что путь может и не иметь дуги, так что состояние *0* также достигается из состояния *0*, а значит, входит в ис-комое множество.

С помощью алгоритма на рис. 2.8 можно пометить *A* и вычислить *ε-closure(move(A, a))*. Вычислим *move(A, a)*, множество состояний *N*, имеющих переходы по символу *a* для элементов множества *A*. Из состояний 0, 1, 2, 4 и 7 только 2 и 7 имеют такие переходы к состояниям 3 и 8, поэтому *ε-closure(move({0,1,2,4,7},a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8}*. Назовем это множество *B*. Таким образом, *Dtran[A, a] = B*.

Среди состояний в *A* только состояние 4 имеет переход по *b* в состояние 5, так что ДКА имеет переход по *b* из *A* в состояние *C = ε-closure({5}) = {1,2,4,5,6,7}*. Таким образом, *Dtran[A, b] = C*.

Если продолжить этот процесс с не помеченными в настоящий момент множествами *B* и *C*, в конечном итоге все множества-состояния ДКА окажутся помеченными. Поскольку всего имеется 2^{11} различных подмножеств множества из одиннадцати состояний, а однажды помеченное множество остается таковым навсегда. В действительности мы создаем пять различных множеств состояний:

A = {0, 1, 2, 4, 7}
B = {1, 2, 3, 4, 6, 7, 8}
C = {1, 2, 4, 5, 6 , 7}
D = {1, 2, 4, 5, 6 , 7, 9}
E = {1, 2, 4, 5, 6, 7, 10}

Состояние *A* является начальным, а *E* — единственным заключительным состоянием.

Полностью таблица переходов *Dtran* показана на рис. 2.11.

Состояние	Входной символ	
	a	*b*
A	*B*	*C*
B	*B*	*D*
C	*B*	*C*
D	*B*	*E*
E	*B*	*C*

Рис.2.11. Таблица переходов Dtran для ДКА

Граф переходов полученного в результате ДКА показан на рис. 2.12

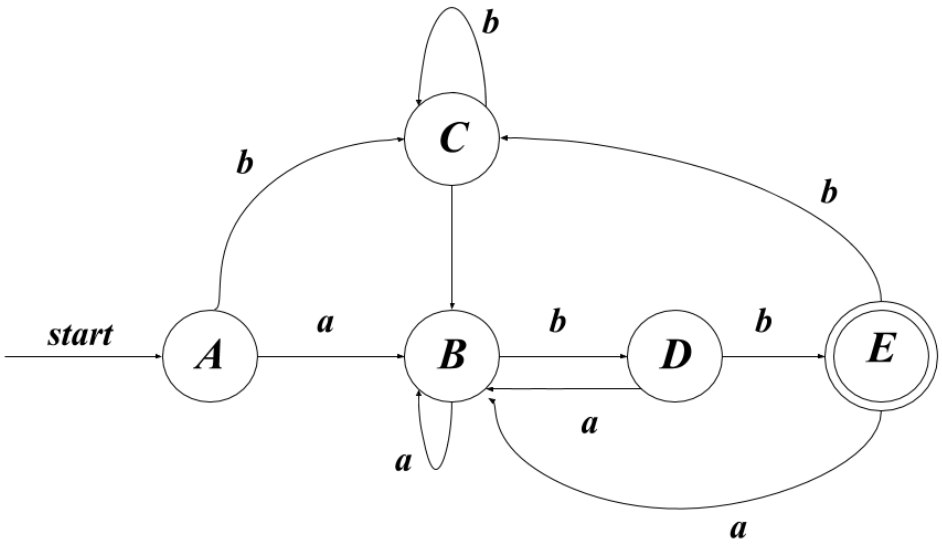


Рис.2.12. Результат применения построения подмножества к рис. 2.10

2.5. Пример проектирования порождающей и распознающей систем

1. Постановка задачи: L = L(G) = L(КА).

Задан язык $L=\{c\omega n \mid \omega \in \{+,d,–,k,f\}^*\}$, где ω обозначает множество всех цепочек, составленных из символов $\{+,d,–,k,f\}^*$. Цепочки начинаются с символа c, заканчиваются символом n.

Спроектировать грамматику $G = (T,V,P,S_0)$ для заданного языка L, привести пример вывода цепочки.

Используя грамматику G, построить КА = $(Q, \sum, \delta, q_0, F)$ и привести пример конфигурации КА. Построить диаграмму переходов КА. Определить свойства КА, если это НКА реализовать алгоритм преобразования НДКА в ДКА.

1.a. Определить свойства языка. Применить лемму о накачке.

1.b. Спроектировать грамматику $G = (T, V, P, S_0)$, где

T = {c, d, k, f, n, +, -} множество терминальных символов

$V = \{S_0, Q\}$ множество нетерминальных символов

$P = \{S_0 \rightarrow Qn, Q \rightarrow c \mid Q+ \mid Q- \mid Qd \mid Qk \mid Qf \}$

1.c. Определить свойства грамматики. Грамматика является леволинейной, бесконечной. Пример вывода цепочки $S_0 \Rightarrow Q\text{-}n \Rightarrow Qf\text{-}n \Rightarrow Qkf\text{-}n \Rightarrow Qdkf\text{-}n \Rightarrow Q\text{-}dkf\text{-}n \Rightarrow Qd\text{-}dkf\text{-}n \Rightarrow Q+d\text{-}dkf\text{-}n \Rightarrow c+d\text{-}dkf\text{-}n$.

1.d. Используя грамматику G построить КА.

$КА= (\{S_0, Q, q_f\}, \{ c, d, k, f, n, +, -\}, \delta, S_0, q_f)$

$\delta(S_0, c) = \{Q\};$

$\delta(Q, d) = \{Q\};\ \delta(Q, k) = \{Q\};\ \delta(Q, f) = \{Q\};\ \delta(Q, +) = \{Q\};\ \delta(Q, -) = \{Q\};$

$\delta(Q, n) = \{q_f\};$

Пример конфигурации КА: $S_0c+d\text{-}dkf\text{-}n \vdash Q+d\text{-}dkf\text{-}n \vdash Qd\text{-}dkf\text{-}n \vdash Q\text{-}dkf\text{-}n \vdash Qdkf\text{-}n \vdash Qkf\text{-}n \vdash Qf\text{-}n \vdash Q\text{-}n \vdash Qn \vdash q_f$

1.e. Определить свойства конечного автомата. Конечный автомат является недетерминированным (НДКА). Преобразовать НДКА в ДКА.

1.f. Построить диаграмму переходов ДКА рис. 2.6.:

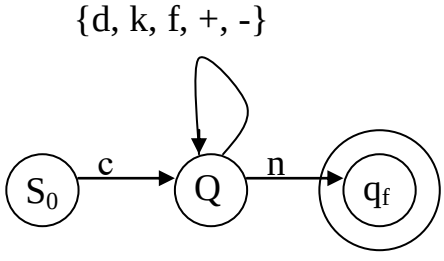


Рис. 2.6. Диаграмма переходов КА

Глава 3. Контекстно-свободные грамматики и МП-автоматы

3.1. Определение КС- грамматик и МП-автоматов

Из четырех типов грамматик иерархии Хомского класс контекстно-свободных грамматик наиболее важен с точки зрения приложений к языкам программирования и компиляции. С помощью этого типа грамматик определяется большая часть синтаксических структур языков программирования.

Определение 6. Контекстно-свободная грамматика типа 2:

Грамматика $G = (T, V, P, S_0)$ называется *контекстно-свободной* (КС) (или *бесконтекстной*), если каждое правило из P имеет вид $A \to \alpha$, где $A \subset V$, $\alpha \subset (T \cup V)^*$.

Заметим, что, согласно определению каждая праволинейная грамматика – КС. Языки, порождаемые КС-грамматиками, называются КС-языками.

Пример, КС-грамматики, порождающей скобочные арифметические выражения: $G_1 = (\{v,+,*,(,)\}, \{S,F,L\}, P, S)$, где $P = \{S \to S + F, S \to F, F \to F * L, F \to L, L \to v, L \to (S)\}$.

Пример вывода, заменяя самый левый нетерминал (левосторонний вывод) сентенциальной формы : $S \Rightarrow S + F \Rightarrow F + F \Rightarrow L + F \Rightarrow v + F \Rightarrow v + F * L \Rightarrow v + L * L \Rightarrow v + v * L \Rightarrow v + v * v$.

3.2. Преобразование КС-грамматик

Для построения синтаксических анализаторов необходимо, чтобы КС-грамматика была в *приведенной* форме см. рис. 3.1.

Если применить к КС-грамматике алгоритмы:

- устранения бесполезных символов

1). *Непроизводящих* и

2). *Недостижимых* символов;

3). преобразования в грамматику без ε–правил;

4). устранения цепных правил, то получим *приведенную* КС-грамматику. Рассмотрим эти алгоритмы и их применение.

1. Символ $a \in T$ называется *недостижимым* в КС-грамматике G, если он не может появиться ни в одной из сентенциальных форм.

2. Нетерминальный символ $A \in V$ называется *производящим*, если из него можно вывести терминальную цепочку, т.е. если существует вывод $A \Rightarrow^+ \alpha$, где $\alpha \in T^+$. В противном случае символ называется непроизводящим.

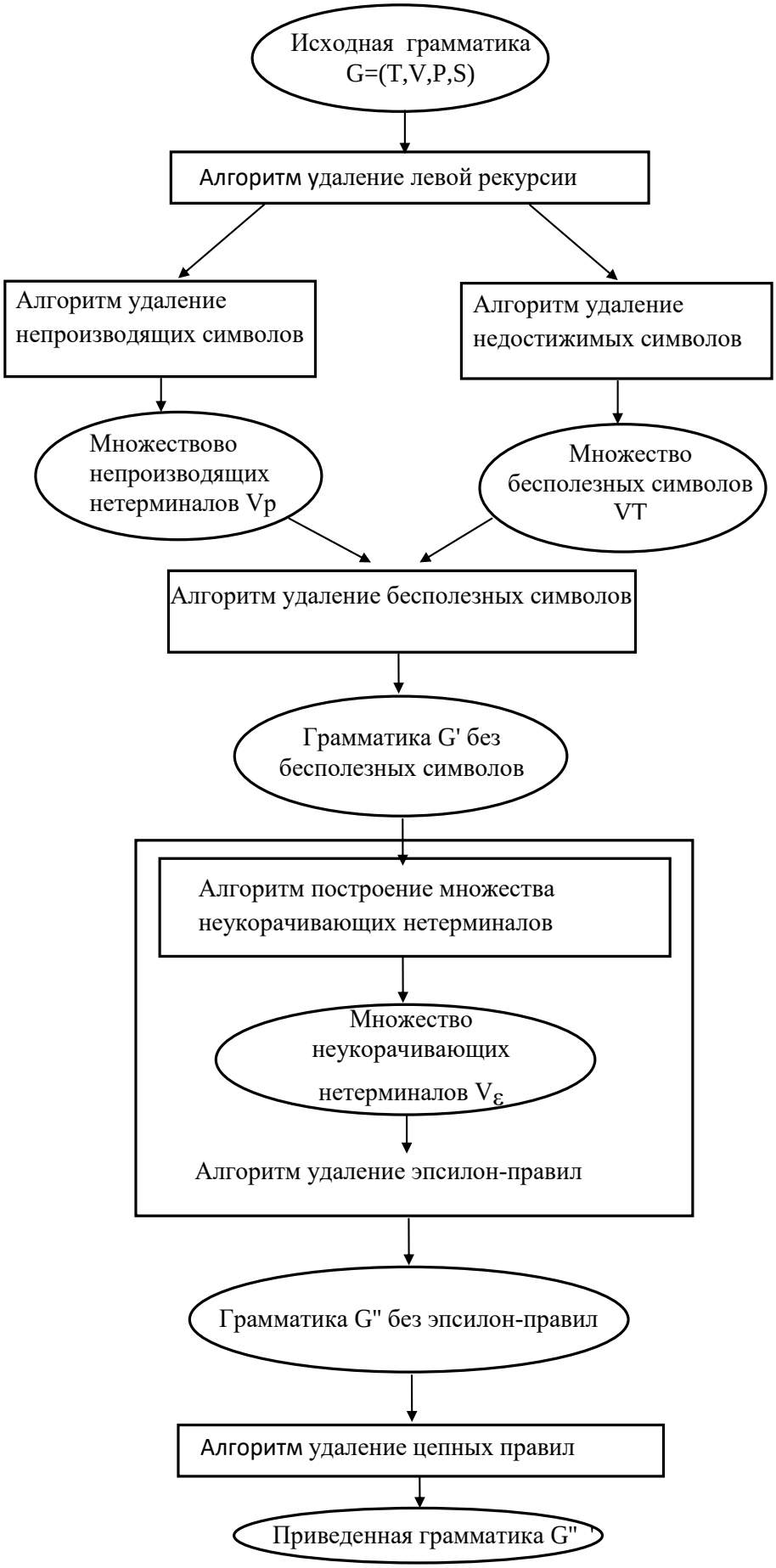


Рис.3.1 Метод преобразования грамматики к *приведенной* форме

Нетерминал КС-грамматики называется *рекурсивным*, если $A \Rightarrow^{+} \alpha A \beta$, для некоторых α и β. Если $\alpha = \varepsilon$, то A называется *леворекурсивным*, если $\beta = \varepsilon$, то A называется *праворекурсивным*. Грамматика, имеющая хотя бы один леворекурсивный нетерминал, называется *леворекурсивной*. Грамматика, имеющая хотя бы один праворекурсивный, нетерминал называется *праворекурсивной*.

При нисходящем синтаксическом анализе требуется, чтобы приведенная грамматика рассматриваемого языка не содержала **левой рекурсии**.

Для любой КС-грамматики существует эквивалентная грамматика без левой рекурсии. Рассмотрим алгоритм устранения левой рекурсии.

Частный случай не леворекурсивной грамматики – грамматика в нормальной форме Шейлы Грейбах.

Определение 7. КС грамматика G = (T, V, P, S) называется грамматикой в нормальной форме Грейбах, если в ней нет ε-правил, т.е. правил вида $A \rightarrow \varepsilon$, и каждое правло из P отличное от $S \rightarrow \varepsilon$, имеет вид $A \rightarrow a\alpha$, где $a \in T$, $\alpha \subset V^{*}$.

Также полезно представлять грамматику в нормальной форме Хомского, что позволяет упростить рассмотрение ее свойств.

Определение 8. КС грамматика G = (T, V, P, S) называется грамматикой в нормальной форме Хомского, если каждое правло из P имеет один из следующих видов:

1. $A \rightarrow BC$, где $A, B, C \in V$;
2. $A \rightarrow a$, где $a \in T$;
3. $S \rightarrow \varepsilon$, если $\varepsilon \in L(G)$, причем S не встречается в правых частях правил.

Алгоритм удаление левой рекурсии.

Вход: КС грамматика $G = (T, V, P, S_0)$.

Выход: Эквивалентная приведенная КС грамматика G'.

1. Расположить нетерминальные символы в некотором порядке. Пусть $V = \{A_1,\ldots,A_n\}$. Преобразуем G так, чтобы в правиле $A_i \rightarrow \alpha$, цепочка α начиналась либо с терминала, либо с такого A_j, что $j > i$. Пусть i=1.

2. Пусть множество A_i правил – это $A_i \rightarrow A_i\alpha_1 |\ldots| A_i\alpha_m | \beta_1 |\ldots| \beta_p$, где ни одна цепочка β_j не начинается с A_k, если $k \le i$. Заменим A_i - правила правилами:

$A_i \rightarrow \beta_1 |\ldots| \beta_p | \beta_1 A_i' |\ldots| \beta_p A_i'$

$A_i' \rightarrow \alpha_1 |\ldots| \alpha_m | \alpha_1 A_i' |\ldots| \alpha_m A_i'$

где A_i' – новый символ. Правые части всех A_i - правил начинаются теперь с терминала или с A_k для некоторого $k > i$.

```
for i := 1 to n
   for  j : = 1 to i -1
      Заменить каждую продукцию вида Ai → Ajα1
       продукциями Ai → β1 |…| βp | β1Ai' |…| βpAi'
                       Ai ' → α1 |…| αm | α1Ai' |…| αmAi'
       где Aj' → α1 |…| αm | α1Ai' |…| αmAi'
   end;
   Устранить непосредсвенную левую рекурсияю среди Ai правил
end;
```

3. Если i = n, то останов и получена грамматика G', иначе j = i, i = i + 1.

4. Заменить каждое правило вида $A_i \to A_j\alpha$ правилами $A_i \to \beta_1\alpha \mid \ldots \mid \beta_m\alpha$, где $A_j \to \beta_1 \mid \ldots \mid \beta_m$ – все A_j - правила.

Так как правая часть каждого A_j – правила начинается уже с терминала или с A_k для k > j, то и правая часть каждого A_i – правила будет обладать этим же свойством.

5. Если j = i - 1, перейти к шагу 2, иначе j = j +1 и перейти к шагу 4.

Пример. Алгоритм удаления левой рекурсии из КС – грамматики G = ({v,+,*,(,)}, {S,F,L}, P, S), где P состоит из правил

$S \to S + F \mid F$

$F \to F * L \mid L$

$L \to v \mid (S)$

Шаг 1 . Пусть $A_1 = S$, $A_2 = F$, $A_3 = L$. $V = \{A_1, A_2, A_3\}$.

Шаг 2. Для i = 1 преобразуем правила: $S \to S + F \mid F$, $\alpha = + F$, $\beta = F$. Заменим S - правила правилами : $S \to F \mid FS'$, и добавим в грамматику правило для нового нетерминала $S' \to + F \mid + FS'$.

Шаг 3. i = 1 и j = 2.

Шаг 4. Для i = 1, j=2 правила вида $A_i \to A_j\alpha$ отсутствуют.

Шаг 2. Для i = 2 преобразуем правила: $F \to F * L \mid L$, $\alpha = * L$, $\beta = L$. Заменим F - правила правилами: $F \to L \mid LF'$, и добавим в грамматику правило для нового нетерминала $F' \to * L \mid LF'$.

Шаг 3. i = 2, j = 2.

Шаг 4. Для i = 2, j =2 правила вида $A_i \to A_j\alpha$ отсутствуют.

Шаг 4. Для i = 3, j =1 правила вида $A_i \to A_j\alpha$ отсутствуют.

Обошли все нетерменалы

Получили правила новой грамматики G':

$S \to F \mid FS'$, $S' \to + F \mid + FS'$, $F \to L \mid LF'$, $F' \to * L \mid LF'$, $L \to (S + F) \mid v \mid (S)$.

Алгоритм удаление непроизводящих символов. Определение множества *производящих* нетерминальных символов V_p.

Если все символы цепочки из правой части правила вывода являются производящими, то нетерминал в левой части правила вывода также должен быть производящим: $V_p = \{A \mid A \Rightarrow^+ \alpha, A \in V, \alpha \in T^+\}$

Вход: КС G = (T, V, P, S)

Выход: $V_p = \{A \mid A \Rightarrow^+ \alpha, A \in V, \alpha \in T^+\}$ // ++

$V^0_p := \varnothing$

$V^i_p := \varnothing$

i = 0

do
$V_p := V^i_p$
i = i + 1;
foreach ($A \to a_1 \dots a_k \dots a_n, a_k \in (T \cup V) \in P$
if($\forall\ k \le n, a_k \in V_p \cup T$) // есть производящее правило
$V_p := V^i_p \cup \{A\}$ // $V^i_p = \{A \mid (A \to \alpha) \in P, \alpha \in (T \cup V^{i-1}_p)^+\}$
end
end
while ($V^i_p \neq V^{i-1}_p$)

Алгоритм удаление недостижимых символов. Определение *множества достижимых символов* VT_r (нетерминальных и терминальных). Если нетерминал в левой части правила грамматика является достижимым, то достижимы и все символы правой части этого правила.

Вход: КС грамматика G = (T, V, P, S).

Выход: $VT_r = \{X \mid S \Rightarrow^+ \alpha X \beta, X \in V \cup T$ и $\alpha, \beta \in (V \cup T)^*\}$.

$VT^0_r := \{S\}$
$VT^{i-1}_r := \varnothing$ // дополнительное множество
i = 0;
do
i = i + 1;
$VT^{i-1}_r = VT^i_r$
foreach ($A \in V$)
if ($A \to \alpha X \beta$) $\in$ P // нетерминал X достижим
$VT^i_r := VT^i_r \cup X$ //$VT^i_r = VT^{i-1}_r \cup \{X \mid A \Rightarrow \alpha X \beta, X \in V \cup T$ и $\alpha, \beta \in (V \cup T)^*$ и $A \in VT^{i-1}_r\}$
if ($\alpha \notin VT^i_r$)
$VT^i_r := VT^i_r \cup \alpha$
end
if ($\beta \notin V^i_r$)
$VT^i_r := VT^i_r \cup \beta$
end
end
end foreach
while($VT^i_r \neq VT^{i-1}_r$)

Алгоритм удаление бесполезных символов. Устранение *бесполезных* символов. Вначале исключить *непроизводящие* нетерминалы, а затем *недостижимые* символы.

Вход: КС грамматика G = (T, V, P, S), для которой $L(G) \neq \varnothing$, $V = V_p$

Выход: КС грамматика G’ = (T’, V’, P’, S), у которой $L(G') \neq \varnothing$ и в $T' \cup V'$ нет бесполезных символов.

Выполнить А**лгоритм удаления непроизводящих символов** и **Алгоритм удаление недостижимых символов** (см. выше).

P' := Ø
foreach (A → αXβ) ∈ P // для каждого правила ∈ P
if X ∈ V_p // X – производящий нетерминал
P' := P' ∪ (A → αXβ) //Добавляем правило, из которого выводится X
end
end foreach
T` := T ∩ VT_r
V` := V ∩ VT_r

Пример: Устранить из грамматики G бесполезные символы. G = (T, V, P, S), где V = {A,B,C,S}, T = {a,b,c}, P = {S→aC, S→A, A→cAB, B→b, C → a}.

множество V_p = {B, C, S} A – *непроизводящий символ*

G_1 = ({B, C, S}, {a,b,c}, P, S), где P = {S→aC, B→b, C→ a},

V_r = {C, S} B – *недостижим, так как* A- *непроизводящий символ*

G' = ({C, S}, {a}, P, S), где P = {S→aC, C → a}, L(G') = {aa}

Описание алгоритма удаление эпсилон-правил. КС-грамматика называется *неукорачивающей* КС-грамматикой (НКС-грамматикой, КС-грамматикой без ε–правил) при условии, что P не содержит S → ε и S не встречается в правах частях остальных правил.

В грамматике с правилами вида A→ε длина выводимой цепочки при переходе от k-го шага к (k+1)-му уменьшается. Поэтому грамматики с правилами вида A→ε называются укорачивающими.

Восходящий синтаксический разбор в укорачивающих грамматиках сложнее по сравнению с разбором в неукорачивающих грамматиках, т.к. при редукции необходимо отыскать такой фрагмент входной цепочки, в которую можно вставить пустой символ.

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

Построение множества V_ε укорачивающих нетерминалов.

Алгоритм построение множества неукорачивающих нетерминалов. Алгоритм устраненеия ε–правил в КС-грамматике основан на использовании множества укорачивающих нетерминалов.

Вход: КС грамматика G = (T, V, P, S).

Выход: V_ε = { A | (A $\Rightarrow^+$ α) ∈ P и α ∈ V_ε^+}.

V^0_ε = Ø
foreach (A ∈ V)
if (A → ε) ∈ P // A → ε правило укорачивающиеся
V_ε := V_ε ∪ A // Положить V_ε = V_ε ∪ { A | (A→ α) ∈ P и α ∈ V_ε^+}
end
end

V^1_ε := V_ε
V^0_ε := Ø

i := 0
do
  i = i + 1
  $V^{i-1}_{\varepsilon} := V^{i}_{\varepsilon}$
  foreach($A \to (C_1 \ldots C_k \ldots C_n, C_k \in V)) \in P$ // для каждого правила $\in P$
    if($\forall k \le n, C_k \in V_{\varepsilon}$)
      $V^{i}_{\varepsilon} := V^{i-1}_{\varepsilon} \cup A$ // Положить $V^{i}_{\varepsilon} = V^{i-1}_{\varepsilon} \cup \{ A \mid (A \to \alpha) \in P$ и $\alpha \in V^{i-1}_{\varepsilon}{}^{+}\}$.
    end
  end
while($V^{i-1}_{\varepsilon} \neq V^{i}_{\varepsilon}$)

Алгоритм удаление эпсилон-правил. Преобразование КС-грамматики с ε–правилами в эквивалентную НКС-грамматику.

Вход: КС G' = (T, V, P, S), V_{ε}
Выход: НКС грамматика G'' = (T, V`, P'', S')

P' := Ø
V' := Ø

foreach ($A \to \alpha_0 B_1 \alpha_1 B_j \alpha_i \ldots B_k \alpha_k$) $\in$ P // для каждого правила $\in$ P
//включить в P', все правила вида $A \to \alpha_0 X_1 \alpha_1 X_j \alpha_i \ldots X_k \alpha_k$, где $X_j = B_j$ или X=ε.
  P' := P' $\cup$ ($A \to \alpha_0 B_1 \alpha_1 B_j \alpha_i \ldots B_k \alpha_k$) // X= ε
  V' := V' $\cup$ A
// Если в правой части нетерминал, из которого выводится ε-правило
  foreach ($B_k \in V_{\varepsilon}$)
    P' := P' $\cup$ ($A \to \alpha_0 B_1 \alpha_1 B_j \alpha_i \ldots B_k \alpha_k \setminus B_k$) // разность
  end
end

if ($S \in V_{\varepsilon}$)
  V' = V' $\cup$ {S'}
  P' := P' $\cup$ ($S' \to S \mid \varepsilon$)
end

Пример: Устранить из грамматика G ε–правила. Преобразовать грамматику G = (T, V, P, S), где V = {A,S}, T = {b,c}, P = {S→cA, S→ ε, A→cA, A→ bA, A→ ε}, в эквивалентную НКС-грамматику.

Шаг 1. Применяя **алгоритм построения множества неукорачивающих нетерминалов**, получаем $V^{0}_{\varepsilon} = \{S, A\}$, $V^{1}_{\varepsilon} = \{S, A\}$, значит $V^{0}_{\varepsilon} = V^{1}_{\varepsilon} = \{S, A\}$.

Шаг 2. Положить P' = Ø.

Рассмотрим правило S→cA.

Для него в новое множество правил грамматики P' добавляем правила S→cA и S→c.

Для правила A→cA, добавляем в P' правила A→cA и A→c.

Для правила A→bA, добавляем в P' A→bA и A→b, тогда P' = {S→cA, S→c, A→cA, A→c, A→bA, A→b}.

Шаг 3. $S \in V_\varepsilon$, то в V' добавляем новый нетерминал S', а в P' два правила $S' \to S$ и $S' \to \varepsilon$.

G' = ({b,c}, { S', S, A} , { $S' \to S$, $S' \to \varepsilon$, $S \to cA$, $S \to c$, $A \to cA$, $A \to c$, $A \to bA$, $A \to b$}, S').

Алгоритм удаление цепных правил.

Правило вида $A \to B$, где A и B — нетерминалы называется цепным.

Вход: КС грамматика G = (T, V, P, S).

Выход: Эквивалентная НКС грамматика G' = (T, V, P', S) без цепных правил.

$P_c := \varnothing$ // множество цепных правил

$P_{nc} := \varnothing$ //множество нецепных правл

Шаг 1. Нахождение цепных и нецепных правил

```
foreach (A → αBβ) ∈ P
   if (B ∈ V  &  α,β = Ø)  // Если  A → B  - цепное
      Pc := Pc ∪ (A → B)  // добавляем его в мн-во цепных Pc
   else
      Pnc := Pnc ∪ (A → αBβ)  // иначе добавляем в мн-во нецепных Pnc
   end
end
P' := Ø
```

Шаг 2. Замена цепных на нецепные

```
foreach (R→ αBβ) ∈ Pnc
  if (B→ C) ∈ Pc
    P' := P' ∪ (R→ αCβ)
 else
    P' := P' ∪ (R→ αBβ)
 end
end
```

Пример Устранить из КС грамматики G цепные правила, G = ({S,F,L}, {v,+,*,(,)}, P, S), где P состоит из правил

$S \to S + F \mid F$, $F \to F * L \mid L$, $L \to v \mid (S)$

Шаг 1. Получаем $V_S = \{S, F, L\}$, $V_F = \{F, L\}$, $V_L = \{L\}$.

Положить P' = Ø.

Шаг 2. Выберем первый нетерминал B из множества V. Множество правил, левая часть которых – нетерминальнй символ R, правые части - это правые части нецепных правил исходной грамматики, в левой части которых находятся символы из множества V , получаем: {$S \to S + F \mid F*L \mid (S) \mid v$ }. Включаем эти правила в P'={$S \to S + F \mid F*L \mid (S) \mid v$ }. Таким же образом рассматриваем символы из V_F , V_L , В результате G' = (T, V, P', S), где P'={$S \to S + F \mid F*L \mid (S) \mid v$, $F \to F * L \mid (S) \mid v$, $L \to v \mid (S)$}

Интегрированный алгоритм (Курсовая)

Данные алгоритмы можно интегрировать, объединяя похожие части. Интегрируем алгоритмы 1-2:

$VT_r := \{S_0\}$
$V_p := Ø;$
$P' := Ø$
foreach ($A \in V$)
  foreach ($A \to \alpha X \beta) \in P$
    $V_p := V_p \cup A$
    $VT_r := VT_r \cup X$
    if $\alpha \notin VT_r$
      $VT_r := VT_r \cup \alpha$
    end if
    if $\beta \notin VT_r$
      $VT_r := VT_r \cup \beta$
    end if
    if $X \in V_p$
      $P' := P' \cup (A \to \alpha X \beta)$
    end if
  end foreach
end foreach
$T' := T \cap VT_r$
$V' := V \cap VT_r$

Таким же способом можно объединить и другие алгоритмы, но их сложность, как и в алгоритме выше, повысится, так как чтобы все алгоритмы работали в одном цикле, нужно будет создавать несколько дополнительных внутри него.

Пример применения метода приведения грамматики

Вход: КС грамматика G ({a, b, c, d}, {S, A, B, C, F}, P, S), где P:

$S \to b \mid cAB$
$A \to Ab \mid c$
$B \to cB$
$C \to Ca$
$F \to d$

Результат работы программы:

Правила:

$S \to b$
$S \to A$
$S \to cAB$
$A \to Ab$
$A \to c$
$B \to cB$
$C \to Ca$
$F \to d$

Бесполезные символы удалены...

Правила:
S → b
S → A
S → cAB
A → Ab
A → c
B → cB
Эпсилон-правила удалены...
Правила:
S → b
S → A
S → cAB
S → c
A → Ab
A → b
A → c
B → cB
B → c
Цепные правила удалены...
Правила:
S → b
S → cAB
S → c
A → Ab
A → b
A → c
B → cB
B → c
Левая рекурсия удалена...
A → Ab
Правила:
S → b
S → cAB
S → c
A → b
A → c
B → cB
B → c
A → S0
S0 → bS0

3.3. Свойства КС языков: лемма о накачке

Доказательство леммы накачки для КС языков.

Рассмотрим следующий КС языка в нормальной форме Хомского.

S → AB A → DE B → FD E → FG

G → DB | g F → HG D → d H→ h

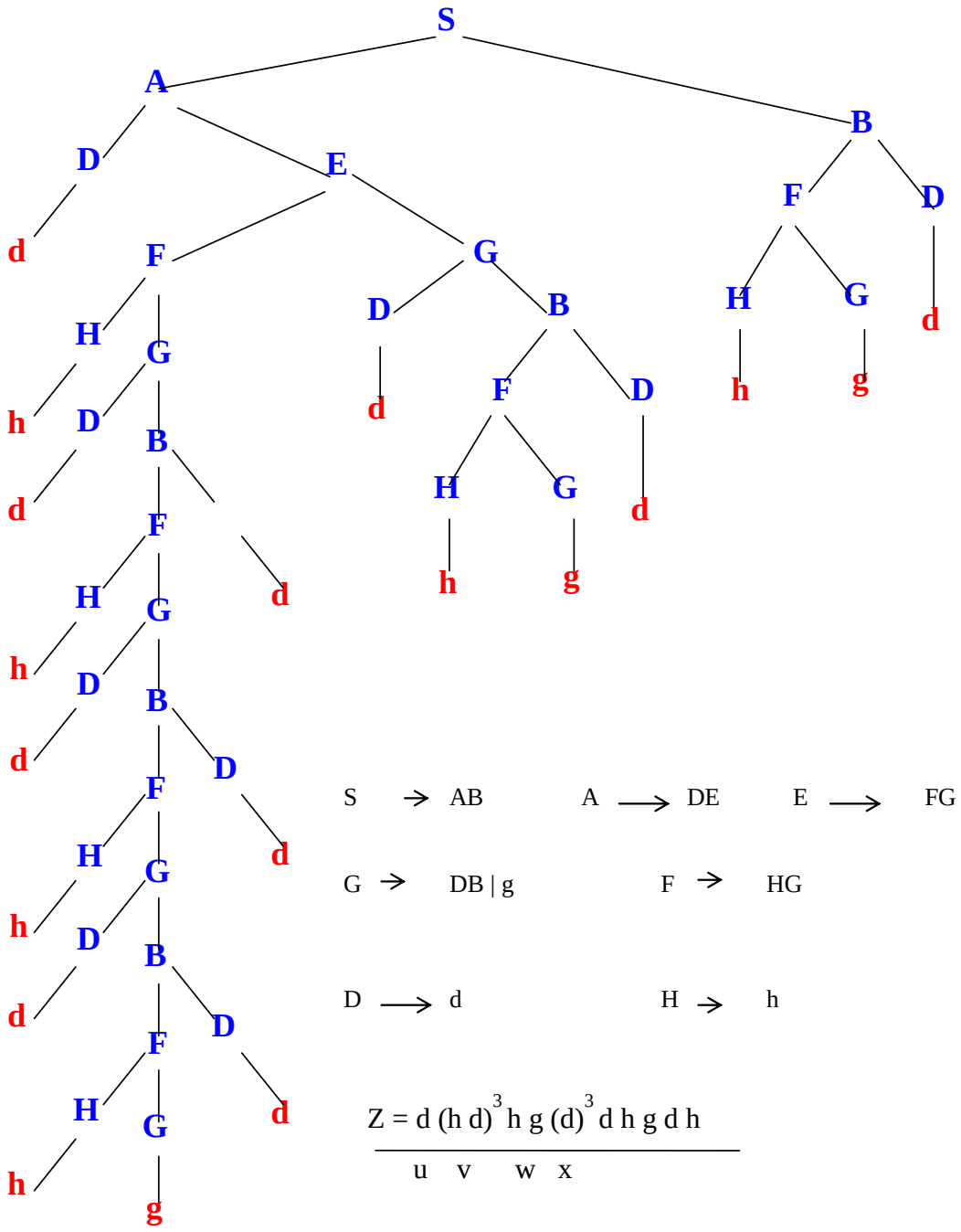


Рис. 2.6. Дерево разбора КС грамматики

Дерево вывода для строки z = d (h d)^3 h g (d)^3 d g h d g h d приведено на рис.1. Обратите внимание на паттерн нетерминальной последовательности FGB-FGB-FGB, которая встречается на самом длинном стволе дерева синтаксического анализа. Он имеет повторяющийся шаблон FGB, который имеет листья “hd” слева и “d” справа. Легко видеть, что мы можем “вырастить” стебель, добавив столько повторяющихся шаблонов F-G-B, сколько мы хотим, или сократить его, удалив шаблон.

Полученное дерево по-прежнему является деревом синтаксического анализа грамматики. Из этого следует, что грамматика имеет дерево разбора для z' = d (h d)^i h g (d)^i d g h d g h d, для всех i ≥ 0, что означает, что z' также находится на языке грамматики.

uvwxy

uvvwxx

uvvvwxxxy

Реализация

Класс **UViWXiY** содержит поля **U,V,i,W,X,Y** соответствующие представлению строки Z = UV^iWX^iY

Класс **PumpingLema** следующие поля и методы:

private string Z - цепочка символов, над которой будет производиться разбор

private List< UViWXiY > ResultZ - результат разбора Z, список структур, в которых содержится результат разбиения **Z** = **UViWXiY** при максимальном возможном значении i

private List<string> WereChecked - список подцепочек **Z**, для которых лема была проверена

private void FindV() - для цепочки Z перебираются все возможные подцепочки, которые принимаются за V

private void FindVi(string V) - для заданной подцепочки проверяется максимальное количество её непрерывных повторений. Если получается превысить полученное до этого число повторений i, результаты разбиения строки **UViWXiY**, результат заменяет предидущий.

private void FindX() - для цепочки Z перебираются все возможные подцепочки, которые принимаются за X

private void FindXi(string X) - для заданной подцепочки проверяется максимальное количество её непрерывных повторений. Если получается превысить полученное до этого число повторений i, результаты разбиения строки **UViWXiY**, результат заменяет предидущий.

public void PrintResult() - выводит на консоль результаты проверки

Цепочки, которые принимаются за V and X выбираются таким образом: во внешнем цикле изменяется длинна, от 1 до половины длинны Z (если длинна V and X превысит половину длины Z, V and X не встретится более 1 раза), во внутреннем цикле выбирается p из Z, в котором будет начинаться получения подцепочка V and X длинны Length V and LengthX соответственно.

```
for (int LengthV = 1; LengthV <= Z.Length / 2; LengthV++){
    for (int p = 0; p < Z.Length - LengthV; ++p{
       this.FindVi(Z.Substring(p, LengthV));
    }
}
for (int LengthX = 1; LengthX <= Z.Length / 2; LengthX++){
    for (int p = 0; p < Z.Length - LengthX; ++p){
      this.FindXi(Z.Substring(p, LengthX));
    }
}
```

Для каждой выбранной подцепочки проверяется максимальное количество её непрерывных повторений.

В начале проверки проверяется, была ли просмотрена подпоследовательность, выбранная в качестве V. Если не была, то заносится в список **WereChecked**, алгоритм продолжается. Цепочку Z можно покрыть V .Lenght способами. Например Для V Длинной 3 Z можно покрыть 3 способами:

Для Каждого из тих способов не сложно подсчитать самое большое количество непрерывных повторений, при этом необходимо запоминать позицию начала соответствующей последовательности повторений. После прохода получившееся максимальное количество повторений сравнивается с предыдущим полученым, и если оно превосходит, предидущие результаты стираются, в стек помещаются новые. Если одному покрытию соответствует несколько одинаковых по количеству повторений последовательностей, в стек добавляются все эти варианты. После завершения проверки для каждого покрытия, в стеке будет содержаться информация разбиении $Z = UV^iWX^iY$, для заданой подцепочки V и максимально возможной i, если таких разбиений несколько - тоже. Результат из стека переносится в **ResultZ** только если значение i больше 1 и не меньше найденого ранее при другом выборе V.

Аналогично приводится пример с подцепочкой X

Пример. Программа корректно выявила разбиения строки $Z = UV^iWX^iY$ с наибольшим возможным значением i.

abcdefedcba

aaaaaaa

a^7

ddddddd

d^7

aaabaaadaaa

aaabaaad a^3

aaab a^3 daaa

a^3 baaadaaa

d^3

abbbbc

a b^4 c

abbbcccddde

a b^3 cccddde

abbb c^3 ddde

abbbccc d^3 e

abcdabcdabcdabcd
abcd^4

aaabcdabcdabcdabcddaab
aa abcd^4 daab

aaabcdbcdbcdaaabcdbcdbcdaaabcdbcdbcd
aaabcdbcdbcdaaabcdbcdbcd a^3 bcdbcdbcd
aaabcdbcdbcd a^3 bcdbcdbcdaaabcdbcdbcd
a^3 bcdbcdbcdaaabcdbc
d^3 dbcdaaabcdbcdbcd
aaabcdbc d^3
dbcdaaabcdbcdbcdaaa bcd^3
aaabcdbcdbcdaaa bcd^3 aaabcdbcdbcd
aaa bcd^3 aaabcdbcd d^3 dcdaaabcdbcdbcd
aaabcdbcdbcd^3

abbbbabbbbabbbbabbbb
abbbbabb d^3 bbabbbba b^4
abbbbabbbba b^4 abbbb
abbbba b^4 abbbbabbbb
a b^4 abbbbab d^3 bbbabbbb
abbbb^4

abcdbcdbcdbcdbcd
a bcd^5
abcbcbcbd
a bc^3 bd
ab cb^3 d d^3 cd

3.4. Определение МП-автоматов

Автоматы с магазинной памятью (МП - автоматы) представляют собой модель распознавателей для языков, задаваемых КС-грамматиками. МП -

автоматы имеют вспомогательную память, называемую магазином см. 1.10. В магазин можно поместить неограниченное количество символов. В каждый момент времени доступен только верхний символ магазина.

Верхний символом магазина будем считать самый левый символ цепочки.

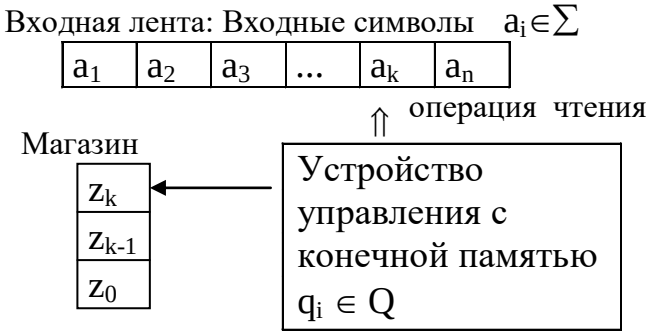


Рис. 1.10. Модель МП - автомата

Определение 9. МП автомат – это семерка объектов

МП = $(Q, \sum, \Gamma, \delta, q_0, z_0, F)$

Q – конечное множество состояний устройства управления;

$\sum$ - конечный алфавит входных символов;

Γ - конечный алфавит магазинных символов;

δ - функция переходов, отображает множества $Q\times(\sum \cup \{\varepsilon\})\times\Gamma$ в множество конечных подмножеств множества $Q\times\Gamma^*$;

q_0 - начальное состояние, $q_0 \in Q$;

z_0 - начальный символ магазина, $z_0 \in \Gamma$;

F - множество заключительных состояний, $F \subseteq Q$.

Определение 10. Конфигурацией МП-автомата называется тройка $(q, \omega, z) \in Q\times\sum^*\times\Gamma^*$, где

q – текущее состояние управляющего устройства;

ω – необработанная часть входной цепочки (первый символ цепочки ω находится под входной головкой; если $\omega = \varepsilon$, то считается, что вся входная цепочка прочитана) ;

z – содержимое магазина (самый левый символ цепочки z считается верхним символом магазина; если $z = \varepsilon$, то магазин считается пустым).

Такт МП-автомата будем описывать бинарным отношением $\vdash$, определенным на множестве конфигураций. Будем писать:

$(q, a\omega, z) \vdash (q', \omega, z\gamma)$, если $\delta(q, a, z) = (q', \gamma)$, где

$q, q' \in Q$, $a \in \sum \cup \{\varepsilon\}$, $\omega \in \sum^*$, $z \in \Gamma$ и $\gamma \in \Gamma^*$

Если $a \neq \varepsilon$, то и входная цепочка прочитана не вся, то запись $(q, a\omega, z\gamma) \vdash (q', \omega, a\gamma)$ означает, что МП-автомат в состоянии q, обозревая символ во входной цепочки и имея символ z в верхушке магазина, может перейти в новое состояние q', сдвинуть входную головку на один символ вправо и заменить верхний символ магазина z цепочкой магазинных символов γ.

Если z = a, то верхний символ удаляется из магазина.

Если $a = \varepsilon$, то текущий входной символ в этом такте называется ε-таком, не принимается во внимание и входная головка остается неподвижной.

ε-такты могут выполняться также в случае, когда вся входная цепочка прочитана, но если магазин пуст, то такт МП-автомата невозможен по определению.

Так же, как и для конечных автоматов, можно определить транзитивное $\vdash^+$ и рефлексивно-транзитивное $\vdash^*$ замыкания (отношения $\vdash$).

Начальной конфигурацией МП-автомата называется конфигурация вида (q_0, ω, z_0), где устройство управления находится в начальном состоянии, на входной ленте записана цепочка $\omega \in \Sigma^*$, которую необходимо распознать, а магазин содержит только начальный символ z_0.

Заключительной конфигурацией МП-автомата называется конфигурация вида (q, ε, γ), где $q \in F$ – одно из заключительных состояний устройства управления, входная цепочка прочитана до конца, а в магазине записана некоторая, заранее определенная цепочка $\gamma \in \Gamma^*$.

Есть два способа определить язык, допускаемый МП-автоматом:

1. множеством входных цепочек, для которых существует опустошающая магазин последовательность операций;

2. множеством входных цепочек, для которых существует последовательность операций, приводящая автомат в заключительное состояние.

Цепочка $\omega \in \Sigma^*$ допускается МП-автоматом МП =(Q, Σ, Γ, δ, q_0, z_0, F), если $(q_0, \omega, z_0) \vdash^* (q, \varepsilon, \gamma)$ для некоторых $q \in F$ и $\gamma \in \Gamma^*$.

Язык распознаваемый МП-автоматом, называется множество цепочек:

$L(МП) = \{\omega \mid \omega \in \Sigma^*$ и $(q_0, \omega, z_0) \vdash^* (q, \varepsilon, \gamma)$ для некоторых $q \in F$ и $\gamma \in \Gamma^*\}$

Пример. Определим МП-автомат, допускающий язык $L = \{a^n b^n \mid n \geq 0\}$

Цепочка символов языка L: aaabbb

МП = ({q_0, q_1, q_2, q_f}, {a,b}, { z_0,a}, δ, q_0, z_0, { q_f })

$\delta(q_0,a, z_0) = \{(q_1,az_0)\}$

$\delta(q_1,a, a) = \{(q_1,aa)\}$

$\delta(q_1,b, a) = \{(q_2, \varepsilon)\}$

$\delta(q_2,b, a) = \{(q_2, \varepsilon)\}$

$\delta(q_2, \varepsilon, z_0) = \{(q_f, \varepsilon)\}$

Последовательность тактов:

$(q_0, aaabbb, z_0) \vdash_1 (q_1, aabbb, az_0) \vdash_2 (q_1, abbb, aaz_0) \vdash_2$
$(q_1, bbb, aaaz_0) \vdash_3 (q_2, bb, aaz_0) \vdash_4 (q_2, b, az_0) \vdash_4$
$(q_2, \varepsilon, z_0) \vdash_5 (q_f, \varepsilon, \varepsilon)$

Алгоритм 3.8. По КС-грамматике G = (T, V, P, S) можно построить МП автомат, L(МП) = L(G). Пусть МП = ({q}, Σ, $\Sigma \cup V$, δ, q, S, {q}), где δ определяется следующим образом:

1. Если $A \to \alpha$ - правило грамматики G, то $\delta(q, \varepsilon, A) = (q,\alpha)$.

2. $\delta(q, a, a) = \{(q, \varepsilon)\}$ для всех $a \in \Sigma$.

Пример проектирования МП-автомата. Постановка задачи. Построить МП-автомат и расширенный МП-автомат по КС-грамматики G = (T, V, P, S),

без левой рекурсии, автомат распознает скобочные выражения. Написать последовательность тактов для выделенной цепочки. Определить свойства автомата.

1. А). Построить МП-автомат, распознающий скобочные арифметические выражения заданные КС-грамматикой (не приведенная) G_1 = ({v,+,*,(,)}, {S,F,L}, P, S), где P состоит из правил: S → S + F | F, F → F * L | L, L → v | (S).

Используя алгоритм 3.8. получим: МП = ({q}, {v,+,*,(,)}, {v,+,*,(,), S,F,L}, δ, q_0, S, {q}), в котором функция переходов δ определяется следующим образом:

1. δ(q_0, ε, S) = {(q, S + F), (q, F)};
2. δ(q, ε, F) = {(q, F * L), (q, L)};
3. δ(q, ε, L) = {(q, v), (q, (S))};
4. δ(q, a, a) = {(q, ε)} для всех a∈∑={v,+,*,(,)}.

B). Последовательность тактов МП-автомата для цепочки v*(v+v) :

(q_0, v * (v+ v), S)⊢1 (q, v * (v+ v), F)⊢ (q, v * (v+ v), F*L)⊢(q, v * (v+ v), L*L) ⊢(q, v * (v+ v), v*L) ⊢ (q, * (v+v), *L) ⊢ (q, (v+v), L)⊢ (q, (v+v), (S)) ⊢ (q, v+v), S)) ⊢ (q, v+v), S+F))⊢(q, v+v), F+F)) ⊢(q, v+v), L+F)) ⊢(q, v+v), v+F)) ⊢(q, +v), +F)) ⊢(q, v), F)) ⊢(q, v), L)) ⊢(q, v), v)) ⊢ (q,),)) ⊢ (q, ε, ε).

Последовательность тактов, которую выполняет МП-автомат, соответствует левому выводу цепочки v * (v+v) в грамматике G_1.

Тип синтаксических анализаторов, которые можно построить таким образом называют *нисходящим* (предсказывающим) анализатором.

Синтаксические анализаторы, построенные на основе расширенного МП-автомата, называют *восходящими* анализаторами, вывод строится снизу в верх.

Определение 11. Расширенным РМП-автоматом называется семерка объектов РМП = (Q, ∑, Г, δ, q_0, z_0, F), где верхним элементом магазина является самый правый символ цепочки, δ-функция переходов, которая задает отображение множества Q×(∑ ∪ {ε})×Г в множество конечных подмножеств множества Q×$Г^*$, а все остальные объекты такие же, как и у МП-автомата см. определение 9.

Конфигурация расширенного МП-автомата определяется так же, как и для МП-автомата.

Алгоритм 3.9. По КС-грамматике G = (T, V, P, S) можно построить расширенный РМП автомат, L(РМП)= L(G). Будем обозначать символом ⊥ - конец цепочки. Пусть РМП = ({q,r}, ∑, ∑ ∪ V ∪ ⊥, δ, q, ⊥, {r}), где δ определяется следующим образом:

1. δ(q, a, ε) = {(q, a)} для всех a∈∑ (символы с входной ленты на этих тактах переносятся в магазин).
2. Если A → α - правило вывода грамматики G, то δ(q, ε, α) = (q, A).
3. δ(q, ε, ⊥S) = {(r, ε)}

РМП-автомат продолжает работу пока магазин не станет пустым.

На практике часто используются детерминированные МП-автоматы.

Пример построения расширенного РМП-автомата

А). Построение расширенного РМП-автомата (Q, ∑, Г, δ, q_0, z_0, F), используем алгоритм 3.9, получим РМП = ({q,r}, {v,+,*,(,)}, {v,+,*,(,), S,F,L,⊥}, δ, q, ⊥, {r}), где функция переходов δ определена следующим образом:

1. δ(q, a, ε) = {(q, a} для всех a∈∑={v,+,*,(,)};
2. δ(q, ε, S+F) = {(q, S)};
3. δ(q, ε, F) = {(q, S)};
4. δ(q, ε, F * L) = {(q, F)};
5. δ(q, ε, L) = {(q, F)};
6. δ(q, ε, v) = {(q, L)};
7. δ(q, ε, (S)) = {(q, L)};
8. δ(q, ε, ⊥S) = {(r, ε)};

В). При анализе входной цепочки v+v*v расширенный РМП-автомат выполнит следующую последовательность тактов:

(q, v+v*v, ⊥) $\vdash^{1}$ (q,+v*v, ⊥v)
$\vdash^{6}$ (q, +v*v, ⊥L)
$\vdash^{5}$ (q, +v*v, ⊥F)
$\vdash^{3}$ (q, +v*v, ⊥S)
$\vdash^{1}$(q, v*v, ⊥S+)
$\vdash^{1}$ (q, *v, ⊥S+v)
$\vdash^{6}$ (q, *v, ⊥S+L)
$\vdash^{5}$ (q, *v, ⊥S+ F)
$\vdash^{1}$ (q, v, ⊥S+ F*)
$\vdash^{1}$ (q, ε, ⊥S+ F*v)
$\vdash^{6}$ (q, ε, ⊥S+ F*L)
$\vdash^{4}$ (q, ε, ⊥S+F)
$\vdash^{2}$ (q, ε, ⊥S)
$\vdash^{8}$ (r, ε, ε)

МП-автомат и расширенный РМП-автоматы – детерминированные автоматы.

Обычно синтаксический анализ выполняется путем моделирования МП-автомата, анализирующего входные цепочки.

МП-автомат отображает входные цепочки в соответствующие левые выводы (нисходящий анализ) или правые разборы (восходящий анализ).

Пусть задана КС-грамматика G = (T, V, P, S), правила которой пронумерованы числами 1, 2, …, p, и цепочка $\alpha \in (T \cup V)^*$,

- левым разбором цепочки α называется последовательность правил, примененных при ее левом выводе из S;
- правым разбором цепочки α называется обращение последовательности правил, примененных при ее левом выводе из S.

Определение 12. Детерминированным МП-автоматом (ДМП-автоматом) с магазинной памятью называется МП-автомат ДМП = (Q, ∑, Г, δ, q_0, z_0, F), у которого для каждого q∈Q и z∈Г выполняется одно из условий:

1. δ(q, a, z) содержит не более одного элемента для каждого a∈∑ и δ(q, ε, z) = Ø;

2. δ(q, a, z) = Ø для всех a∈∑, и δ(q, ε, z) содержит не более одного элимента.

Если каждое правило КС-грамматики начинается с терминального символа, причем альтернативы начинаются с различных символов, то достаточно просто построить *детерминированный* синтаксический анализатор.

3.5. Способы реализации синтаксических анализаторов

Рассмотрим три **с**пособа реализации синтаксических анализаторов для заданного языка L(G). Каждый имеет свои достоинства и недостатки. Напомним, что G является приведенной.

1. МП-автомат строим для L(G) = L(MP), правила автомата строятся в соответствии с расссмотренными алгоритмами.

2. Строятся диаграммы Вирта, для каждой грамматики, создается неявный стек при косвеной рекурсси, что не требует реализации.

3. Таблично-управляемый разбор (предварительно строится диаграмма Вирта), диаграмма реализуется в виде списочной структуры, над которой выполняется алгоритм.

Диаграммы Вирта

1. Постановка задачи. Реализовать синтаксический анализатор левым разбором, используя приведенную грамматику без левой рекурсии G = ({a, b, c, -, +, (,)}, { E, T, F, op }, P, S), где P = { E → T op T, op → + | -, T→ F | (E), F→ a | b | c}

2. Шаги реализации.

Выделить цепочку, принадлежащую языку задаваемомому КС-грамматикой G, например: L(G) = a + (b – c).

Привести вывод выделенной цепочки:

E ⇒ T op T ⇒ T op (E) ⇒ F op (T op T) ⇒ a op (T op T) ⇒ a + (T op T) ⇒ a + (F op T) ⇒ a + (b op T) ⇒ a + (b op F) ⇒ a + (b op c) ⇒ a + (b - c)

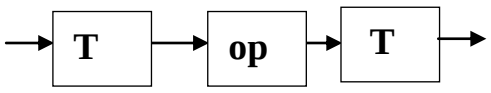
Пронумеровать правила грамматики и придставить их, используя метасимволы {...}, := расширенной формы Бекуса-Наура:

p_1: E := { T op T}, p_2: op := {+,-}, p_3: T := {F , (E)}, p_4: F :={a, b, c}

Составить синтаксический граф, диаграмму Вирта для детерминированного грамматического разбора с просмотром вперед на один символ. Никакие две ветви не должны начинаться с одного и того же символа, если какой-либо граф A можно пройти, не читая вообще никаких входных символов, то такая “нулевая ветвь” должна помечаться всеми символами, которые могут следовать за A (это необходимо для принятия решения о переходе на эту ветвь).

p_1: E:= {T op T}

p_2: op:= {+,-}



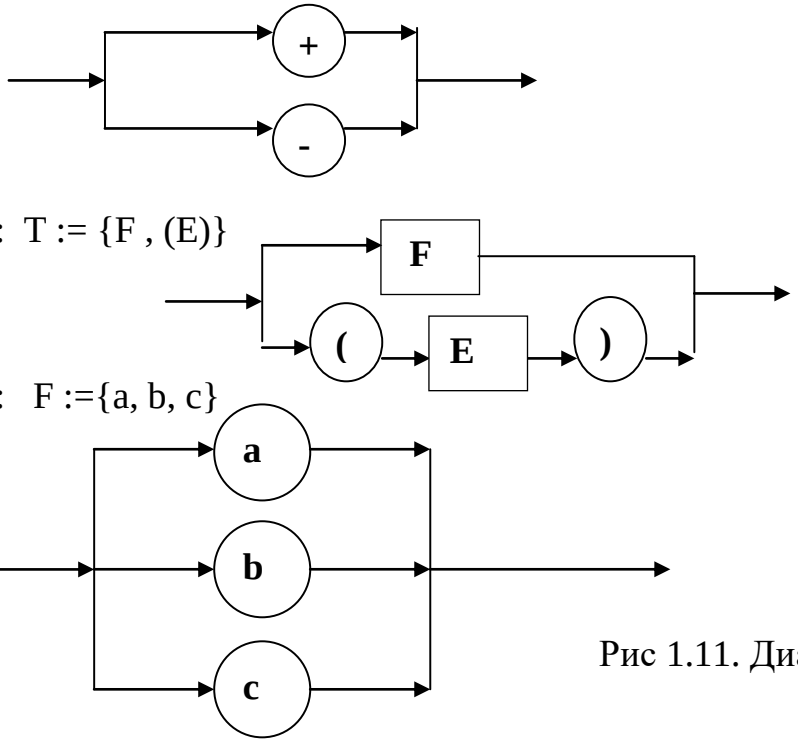


Рис 1.11. Диаграммы Вирта.

Правила преобразования графа в программу:

1. Свести систему графов к возможно меньшему числу отдельных графов с помощью соответствующих подстановок. Сделаем две подстановки, p_2 подставим в p_1, p_4 подставим в в p_3:

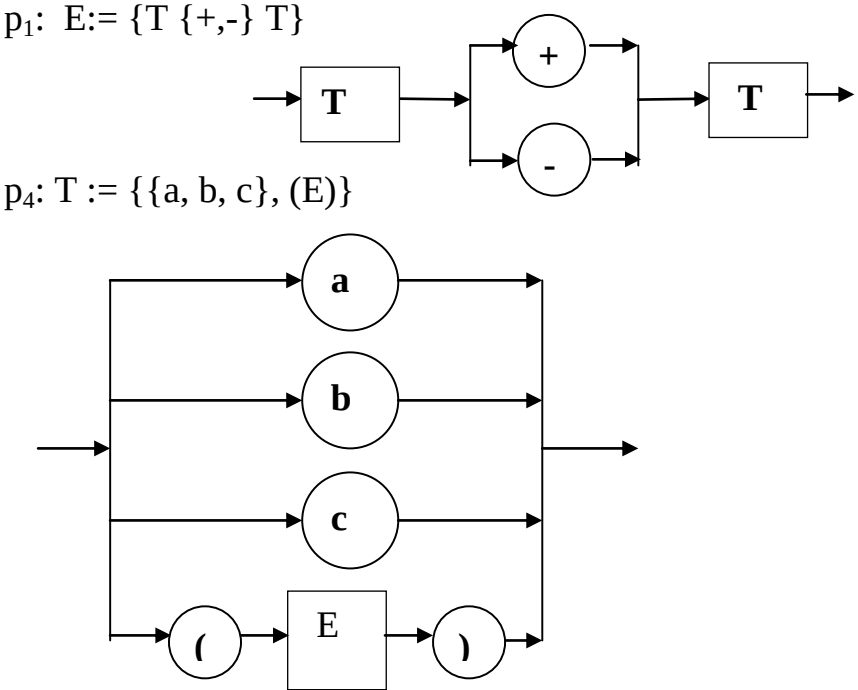


Рис 1.12. Преобразованные диаграммы Вирта.

Полученные правила на графе соответствуют нормальной форме Грейбах.

Преобразовать каждый синтаксический граф в описание процедуры в соответствии с правилами:

1. Каждому графу (диаграмме Вирта) ставится в соответствие процедура.
2. Элемент графа, обозначающий другой граф, переводится в оператор обращения к процедуре.
3. Последовательность элементов в последовательный вызов операторов. Выбор элементов в switch или условный оператор if.

Входные и выходные данные. Пользователь вводит строку в поле редактирования и получает ответ: Да – строка принадлежит языку, порождаемому данной грамматикой, и Нет – строка не принадлежит этому языку. Программа распознает язык L(G) по заданной КС- грамматике G.

Таблично-управляемый разбор

1. Грамматика задается в виде данных, процедура разбора универсально для всех грамматик. Программа работает в строгом соответствии с методом простого нисходящего грамматического разбора и основывается на детерминированном синтаксическом графе (т.е. предложение должно анализироваться просмотром вперед на один символ без возврата).

2. Естественный способ представить граф в виде структуры данных, а не программ – это ввести узел для каждого символа и связать эти узлы с помощью ссылок. Выделим два типа узлов для терминальных (идентифицируется терминальным символом) и нетерминальных символов (ссылка на данные нетерминального символа). Тогда структура узла графически может быть представлена как:

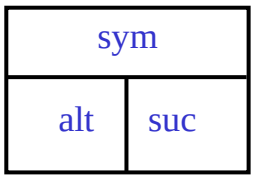


Рис. 1. 14. Структура узла Node

Где “suc” - последователь, “alt” – альтернатива. Пустую последовательность обозначим “empty”. Конструируется класс Node (Узел) для данной структуры.

3. Правила преобразования графов, в структуру данных: получить как можно меньшее число графов с помощью подстановок.

4. Последовательность элементов преобразовать в список узлов (горизонтальный по “suc”), список альтернатив в список узлов (горизонтальный по “alt”), цикл в узел с указателем “suc” на себя, а “alt” на узел с “empty”, где “alt” = NULL, а “suc” выход из цикла.

5. Построим реализацию в виде структуры данных для рассмотренных в п.3 диаграмм.

6. Программа, реализующая структуру на рис. 1.15 приведена ниже. Булевская переменная b управляет алгоритмом разбора. Переменная state определяет состояние цепочки символов.

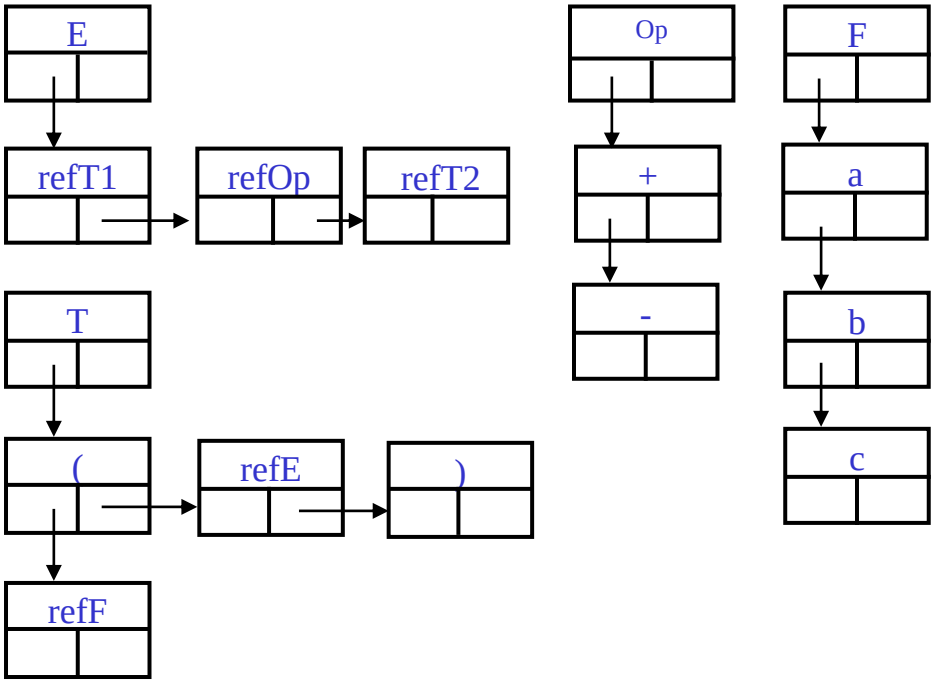


Рис. 1.15. Структура данных для диаграмм п.5

На рис. 1.15. E – начальная вершина Node, а вершины вида refX содержат ссылки на другие вершины. Например, refE содержит ссылку на E, а refT1 и refT2 – на T.

```
   using System;

class Program {
  abstract class State { // абстрактный класс с чисто виртуальной функцией
   public abstract void parse(char c);
   }

// класс для объектов вершин, агрегация по указателю

class Node {
   public Node (char c){ // задание символа вершинам
   this.alt = null;
   this.suc = null;
   this.sym = null;
   this.c = c;
   this.name = c.ToString();
   }
   public Node(string str) {
   this.alt = null;
   this.suc = null;
   this.sym = null;
   this.c = ' ';
   this.name = str;

}
   // метод для соединения вершин
   public void link (Node alt, Node suc, Node sym){
   this.alt = alt;
   this.suc = suc;
   this.sym = sym;
   }
```

```
    public Node sym; // sym != null
    public Node suc; // terminal empty !=' ' && sym == null
    public Node alt;
    public char c;     // empty=' ' && sym == null
    public string name;
    }

    //подкласс для объекта с состоянием правильного разбора
    class OK : State {
    public OK (){}
    public override void parse(char c){
    Console.WriteLine("OK");
    }
    }

//подкласс для объекта с состоянием не правильного разбора
    class ERROR : State {
    public ERROR(){}
    public override void parse(char c) {
    Console.WriteLine("ERROR string");
    }
    }
    //класс для автомата агрегация по ссылке объектов классов OK,
    // ERROR
    class Automate {
    public static State state = null; // полиморфная переменная
    public static ERROR error = new ERROR();
    public static OK ok = new OK();
    public Automate(string str){
    i = 0;
    this.str = str;
    }
    public char getNextChar(){// получить следующий символ из строки
    if (i < str.Length) {
    char ch = str[i];
    i++;
    return ch;
    }
    else {
    return ' ';
    }
    //при окончании строки возвращается пробел
    }
    public virtual void parse(){// начать разбор
    while ( (state !=error)&&(state !=ok) ){
    state.parse(getNextChar()); // принцип подстановки
    }
    state.parse(getNextChar());
    }
    private string str;
    private int i;
    } // end class
    // определение подкласса для объекта автомата, анализирующего
    // контекстно-свободную грамматику
    class AutomateCF : Automate {
    public AutomateCF(string str, Node node):base(str) {
    this.node = node; // передача начальной вершины
    c =' ';
}
    /// <summary>
    /// Проверяет строку, получаемую из getNextChar(), на соответствие данному нетерминалу.
    /// </summary>
    /// <param name="nd">Узел, соответствующий нетерминалу.</param>
    /// <returns></returns>
    public bool parse(Node nd) {
    // Берём первый узел внутри нетерминала.
    Node p = nd.alt;
    // Продолжаем, пока не дойдём до конца нетерминала.
```

```
while (p != null) {
Console.Write(@"Символ строки = '{0}'. Текущий узел = ""{1}"". ", c, p.name);
// Переменная содержит true, если текущий узел соответствует считанной строке.
bool b;
if (p.sym == null) {
// Текущий символ - не ссылка на нетерминал.
if (p.c == c) {
// Текущий символ совпадает со считанным символом.
Console.WriteLine("Совпадение.");
b = true;
c = getNextChar();
state = ok;
// конец строки
if (c == ' ') {
Console.WriteLine("Конец строки.");
return true;
}
} else if  (p.c == ' ') {
// Текущий символ - нетерминал.
Console.WriteLine("Начало нетерминала.");
b = true;
} else {
// Текущий символ - терминал и не совпадает со считанным символом.
Console.WriteLine("Несовпадение.");
b = false;
state = error;
}
}
else {
// Текущий символ - ссылка на нетерминал.
Console.WriteLine(@"Ссылка на нетерминал = ""{0}""", p.sym.name);
// Проверяем, соответствует ли считанная строка текущему нетерминалу.
// Ссылка не может иметь альтернативы, иначе часть считанных символов теряется.
b = parse(p.sym);
}
if (b) {
// Текущий символ соответствует вводу.
if (p.suc != null) {
Console.WriteLine(@"  Переход в suc = ""{0}""", p.suc.name);
}
else {
// Текущий символ - последний символ нетерминала.
Console.WriteLine("  Выход из нетерминала.");
}
p = p.suc;
}
else {
// Текущий символ не соответствует вводу.
if (p.alt != null) {
Console.WriteLine(@"  Переход в alt = ""{0}""", p.alt.name);
}
else {
// Альтернативы нет -> ввод не соответствует нетерминалу.
Console.WriteLine("  Символ не совпадает с узлом, и нет альтернативы.");
return false;
}
p = p.alt;
}
}
// Разбор нетерминала завершился успешно.
return true;
}
public override void parse() { // замещение функции parse в объекте
c = getNextChar();
parse(this.node);
state.parse(getNextChar());
}
```

```
private Node node;
private char c;
}
// программа ввода строки с клавиатуры
static string getString (){ // различные варианты для тестирования..
    // string str = "c+d-dkf-n";
    // string  str = "a+(b-cba)-c";
    // string  str = "a+b";
    // string str = "a+b-c-c+a";
    // string  str = "a+gfg+fgfg"; // error
// string str = "a+(b-c)";
Console.Write("Введите выражение (например, \"a+(b-c)\"): ");
string str = Console.ReadLine();
return str;
}
static void Main(string[] args){
// Построение графа
./ 1. Объявление вершин
// Вершины - начала нетерминалов
// E = {T, op, T}
Node E = new Node("E");
// T = {{(, E, )} | F}
Node T = new Node("T");
// Op = {+ | -}
Node Op = new Node("Op");
// F = {a | b | c}
Node F = new Node("F");
// Вершины - ссылки на нетерминалы
Node refT1 = new Node("refT1");
Node refOp = new Node("refOp");
Node refT2 = new Node("refT2");
Node refE = new Node("refE");
Node refF = new Node("refF");
// Вершины - терминалы
Node braceL = new Node('(');
Node braceR = new Node(')');
Node plus = new Node('+');
Node minus = new Node('-');
Node c_a = new Node('a');
Node c_b = new Node('b');
Node c_c = new Node('c');
// Соединение вершин в дерево методом link
E.link(refT1, null, null);
T.link(braceL, null, null);
Op.link(plus, null, null);
F.link(c_a, null, null);
refT1.link(null, refOp, T);
refOp.link(null, refT2, Op);
refT2.link(null, null, T);
refE.link(null, braceR, E);
refF.link(null, null, F);
braceL.link(refF, refE, null);
braceR.link(null, null, null);
plus.link(minus, null, null);
minus.link(null, null, null);
c_a.link(c_b, null, null);
c_b.link(c_c, null, null);
c_c.link(null, null, null);
//создание объекта автомат
Automate a = new AutomateCF(getString(), E);
//инициализация начальных значений
Automate.state = Automate.ok;
a.parse(); //синтаксический анализ
}
}
```

3.6. LL(k)-грамматики

Синтаксический LL-анализатор - анализирует цепочку символов входного алфавита на ленте слева (L) направо, и строит левый (L) вывод грамматики.

Определение 13. КС-грамматика G = (T, V, P, S) без ε-правил называется простой LL(1) грамматикой (s-грамматикой, разделенной грамматикой), если для каждого v ∈ V все его альтернативы начинаются различными терминальными символами. Единица в названии алгоритма означает, что при чтении анализируемой цепочки, находящейся на входной ленте, входная головка может заглядывать вперед на один символ.

FIRST(A) – это множество первых терминальных символов, которыми начинаются цепочки, выводимые из нетерминала A∈V:

FIRST(A) = $\{a \in T \mid A \Rightarrow^{+}_{i} a\beta \cup \{\varepsilon \text{ if } A \Rightarrow^{*} \varepsilon\}$, где $\beta \in (T \cup V)^{*}\}$

FOLLOW(A) – это множество следующих терминальных символов, которые могут встретиться непосредственно справа от нетерминала в некоторой сентенциальной форме:

FOLLOW(A) = $\{a \in T \mid S \Rightarrow^{*}_{i} \gamma Aa$ и $AaFIRST(\gamma)\}$

Магазин содержит цепочку Xa⊥ (см. рис. 1.17), где Xa – цепочка магазинных символов (X - верхний символ магазина), а символ (⊥) – специальный символ, называемый *маркером дна* магазина. Если верхним символом магазина является *маркер дна*, то магазин пуст. Выходная лента содержит цепочку номеров правил π, представляющую собой текущее состояние левого разбора.

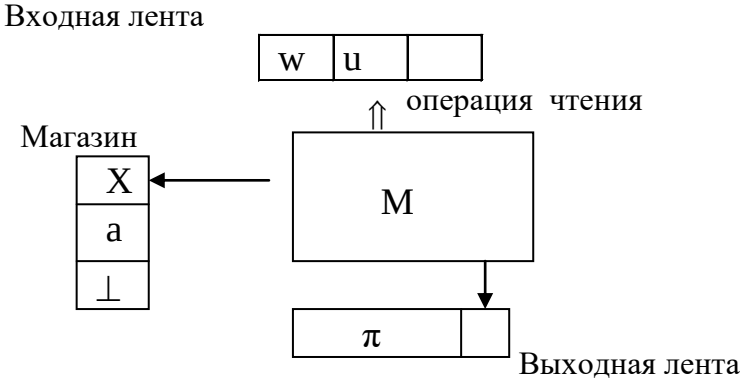


Рис. 1.18. LL(k)-анализатор

Конфигурацию “1-предсказывающего” алгоритма разбора будем представлять в виде (x, Xa⊥, π), где x – неиспользуемая часть входной цепочки, Xa – цепочка в магазине, а π – цепочка на выходной ленте, отражающая состояние алгоритма.

Обозначим алфавит магазинных символов (без символа (⊥)) как V_p.

M – управляющая таблица управляет работой алгоритма. M задает отображение множества $(V_p \cup T \cup \{\perp\}) \times (T \cup \{\varepsilon\})$ в множество, состоящее из следующих элементов:

1. (β, i), где βV_p^{*} правая часть правила вывода с номером i.
2. ВЫБРОС.
3. ДОПУСК.

4. ОШИБКА.

Работа алгоритма в зависимости от элемента управляющей таблицы $M(X,\alpha) = (\beta, i)$ следующая:

1. $(x, X\alpha, \pi) \vdash (x, \beta\alpha, \pi i)$, если $M(X,\alpha) = (\beta, i)$. В этом случае верхний символ магазина X заменяется на цепочку βV_p^*, и в выходную цепочку дописывается номер правила i. Выходная головка при этом не сдвигается.

2. $(ax, a\alpha, \pi) \vdash (x, \alpha, \pi)$, если $M(a,a) =$ ВЫБРОС. Это означает, что, если верхний символ магазина совпадает с текущим входным символом, он выбрасывается из магазина, и входная головка сдвигается на один символ вправо.

3. Если алгоритм достигает конфигурации $(\varepsilon, \perp, \pi)$, что соответствует элементу управляющей таблицы $M(\perp, \varepsilon) =$ ДОПУСК, то его работа прекращается, и выходная цепочка π является левым разбором входной цепочки.

4. Если алгоритм достигает конфигурации $(x, X\alpha, \pi)$ и $M(X,a) =$ ОШИБКА, то разбор прекращается и выдается сообщение об ошибке.

Конфигурация $(\omega, S\perp, \varepsilon)$, где $S \in V_p$ – начальный символ магазина (начальный символ грамматики), называется *начальной конфигурацией.*

Если $(\omega, S\perp, \varepsilon) \vdash^+ (\varepsilon, \perp, \pi)$, то π называется выходом алгоритма для входа ω.

Алгоритм 3.10. Построение управляющей таблицы M для LL(1)-грамматики

Вход: LL(1)-грамматика G = (T, V, P, S)

Выход: Управляющая таблица M для грамматики G.

Таблица M определяется на множестве $(V \cup T \cup \{\perp\}) \times (T \cup \{\varepsilon\})$ по правилам:

1. Если $A \to \beta$ – правило грамматики с номером i, то $M(A, a) = (\beta, i)$ для всех $a \neq \varepsilon$, принадлежащих множеству FIRST(A).

Если $\varepsilon \in$ FIRST(β), то $M(A, b) = (\beta, i)$ для всех $b \in$ FOLLOW(A).

2. M(a, a) = ВЫБРОС для всех $a \in T$.

3. $M(\perp, \varepsilon) =$ ДОПУСК.

4. В остальных случаях M(X, a) = ОШИБКА для $X(V \cup T \cup \{\perp\})$ и $a \in T\{\varepsilon\}$

Естественным обобщением LL(1)-грамматик являются LL(k)-грамматики. Для КС-грамматика G = (T, V, P, S) определим множество:

$FIRST_k(\alpha) = \{x \mid \alpha \Rightarrow^*_i x\beta$ и $|x| = k$ или $\alpha \Rightarrow^* x$ и $|x| < k\}$

Применение алгоритма 3.10. требует использование аванцепочек длиной до k символов, что существенно увеличивает размеры управляющей таблицы.

Кроме того, для некоторых LL(k)-грамматик (k > 1) верхний символ магазина и аванцепочка длиной k (или меньше) символов не всегда однозначно определяют правило, которое должно быть определено при разборе.

Если в КС-грамматике G = (T, V, P, S) для двух различных A-правил $A \to \beta$ и $A \to \gamma$ выполняется:

$FIRST_k(\beta\ FOLLOW_k(A)) \cap FIRST_k(\gamma\ FOLLOW_k(A)) = \varnothing$, то такая

КС-грамматика называется *сильно* LL(k)-грамматикой.

Сложность построения синтаксических анализаторов и их неэффективность не позволяют практически использовать LL(k)-грамматики.

Проектирование LL(k)-анализатора.

A) Построить управляющую таблицу M для грамматики G = (T, V, P, S), где P = {p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8}. p_1:S→ TE’, p_2: E’→ +TE’, p_3: E’→ ε, p_4: T→ PT’, p_5: T’→ *PT’ , p_6: T’→ ε , p_7: P→ (S), p_8: P→ i

B) Рассмотреть работу алгоритма для выделенной цепочки.

C) Определить является ли LL(k)-грамматика *сильно* LL(k)-грамматикой.

A) Используем алгоритм 3.10. Управляющая таблица должна содержать 11 строк, помеченных символами из множества (V ∪ T ∪ {⊥}), и 6 столбцов, помеченных символами из множества (T ∪ {ε}).

	i	(	)	+	*	ε
S	TE’, 1	TE’, 1				
E’			ε, 3	+TE’, 2		ε, 3
T	PT’,4	PT’,4				
T’			ε, 6	ε, 6	*PT’,5	ε, 6
P	i, 8	(S),7				
i	ВЫБРОС					
(		ВЫБРОС				
)			ВЫБРОС			
+				ВЫБРОС		
*					ВЫБРОС	
⊥						ДОПУСК

Шаг 1. Строим таблицу построчно. Последовательно рассмотрим все нетерминальные символы.

1. Нетерминалу S соответствует правило вывода грамматики p_1: S → TE’. Так как FIRST(TE’) = { (, i }, два терминальных символа, то:

M(S, ( ) = M(S, i) = TE’, 1

2. Для нетерминала E’ в грамматике имеются два правила вывода p_2 и p_3:

p_2: E’→ +TE’ множество FIRST(+TE’) = {+} и, следовательно, M(E’, +) = +TE', 2 ;

p_3: E’→ ε имеет простую правую часть, вычислим множество символов, следующих за нетерминалом E’ в сентенциальных формах. Построив левый вывод S ⇒ TE’ ⇒ PT’E’ ⇒ (S)T’E’ ⇒ (TE’)TE’ …, имеем

FOLLOW(E’) = {), ε}.

Таким образом, M(E’,)) = M(E', ε) = (ε, 3).

Выполняя шаг 1 алгоритма для нетерминалов T, T’ и P получим:

Правило грамматики	Множество	Значение M
p_4: T→ PT’ p_5: T’→ *PT’ p_6: T’→ ε p_7: P→ (S) p_8: P→ i	FIRST(PT’) = {(,i} FIRST(*PT’) = {*} FOLLOW(T’)= {+,), ε} FIRST((S)) = {(} FIRST(i) = {i}	M(T’,()=M(T’,i) = TE’,4 M(T’,*)=*PT’,5 M(T’,+)=M(T’,))=M(T’,ε)=ε, 6 M(P,() =(S),7 M(P,i) =i,8

Шаг 2. Далее всем элементам таблицы, находящимся на пересечении строки и столбца, отмеченных одним и тем же терминальным символом, присвоим значение ВЫБРОС.

Шаг 3. Элементу таблицы M(⊥, ε) присвоим значение ДОПУСК.

Шаг 4. Остальным элементам таблицы присвоим значение ОШИБКА и представим результат в виде таблицы.

Начальное содержимое магазина - S⊥

B) Рассмотрим работу алгоритма для цепочки i+i*i.

Шаг 1. Алгоритм находится в начальной конфигурации (i+i*i, S⊥, ε). Значение управляющей таблицы M(S, i) = TE’, 1, при этом выполняются следующие действия:

- заменить верхний символ магазина S цепочкой TE’;
- не сдвигать читающую головку;
- на выходную ленту поместить номер использованного правила 1.

Шаг 2. Выполняя действия, аналогичные описанным для шага 1, получим следующие конфигурации:

Текущая конфигурация	Значение M
(i+i*i, TE’⊥, 1) ⊢ (i+i*i, PT’E’⊥, 14) ⊢ (i+i*i, iT’E’⊥, 148) ⊢	M(T, i) = PT’, 4 M(P,i) = i,8 M(i,i) = ВЫБРОС

Шаг 3. Алгоритм находится в конфигурации (i+i*i, iT’E’⊥, 148). M(i,i)=ВЫБРОС. Выполняем следующие действия:

- удаляем верхний символ магазина;
- сдвигаем читающую головку на один символ вправо;
- при этом выходная лента не изменяется.

Алгоритм переходит в конфигурацию (+i*i, T’E’⊥, 148).

Выполняя шаги 1 и 2, алгоритм произведет следующие действия:

Текущая конфигурация	Значение M
(+i*i, T’E’⊥, 148) ⊢ (+i*i, E’⊥, 1486) ⊢ (+i*i, +TE’⊥, 14862) ⊢	M(T’, +) = ε, 6 M(E’,+) = + TE’,2

(i*i, TE’⊥, 14862) ⊢ (i*i, PT’E’⊥, 148624) ⊢ (i*i, iT’E’⊥, 1486248) ⊢ (*i, T’E’⊥, 1486248) ⊢ (*i, *PT’E’⊥, 14862485) ⊢ (i, PT’E’⊥, 14862485) ⊢ (i, iT’E’⊥, 148624858) ⊢ (ε, T’E’⊥, 148624858) ⊢ (ε, E’⊥, 1486248586) ⊢ (ε, ⊥, 14862485863)	M(+,+) = ВЫБРОС M(T, i) = PT’, 4 M(P, i) = i, 8 M(i,i) = ВЫБРОС M(T’, *) = *PT’, 5) M(*, *) = ВЫБРОС M(P, i) = (i, 8) M(i,i) = ВЫБРОС M(T’, ε) = (ε, 6) M(E’, ε) = (ε, 3)

Шаг 5. Алгоритм находится в конфигурации (ε, ⊥, 14862485863)

Так как значение M(⊥,ε) = ДОПУСК, то цепочка i+i*i принадлежит языку и последовательность номеров правил 14862485863 на выходной ленте – это ее разбор.

1.7. LR(k)-грамматики

Синтаксический LR-анализатор анализирует входную цепочку слева направо (L), и строит правый (R) вывод грамматики.

Грамматики, для которых можно построить детерминированный восходящий анализатор, называются LR(k)-*грамматиками* (входная цепочка читается слева (Left) направо, выходом анализатора является правый (Right) разбор, k-число символов входной цепочки, на которое можно “заглянуть” вперед для выделения основы).

Для определения LR(k)-*грамматики* используются:

1. Множество $FIRST_k(\gamma)$, состоящее из префиксов длины k терминальных цепочек, выводимых из γ.

Если из γ выводятся терминальные цепочки, длина которых меньше k, то эти цепочки также включаются в множество $FIRST_k(\gamma)$. Формально:

$FIRST_k(\gamma) = \{x \mid \gamma \Rightarrow_l^* xw$ и $|x| = k$ или $\gamma \Rightarrow_l^* x$ и $|x| < k\}$

2. Пополненной грамматикой, полученной из КС-грамматики G = (T, V, P, S), называется грамматики G' = (V∈{S'}, T, P ={S'→ S}, S'). Если правила грамматики G' пронумерованы числами 1,2,..,p то, будем считать, что S' → S – нулевое правило грамматики G', а нумерация остальных правил такая же, как в грамматики G.

Определение 15. КС-грамматика G = (T, V, P, S) называется LR(k)-грамматикой для k ≥ 0, если из существования двух правых выводов для *пополненной* грамматики G' = (T, V', P', S') полученной из G:

$$S' =>_r^* \alpha A w =>_r \alpha\beta w,$$
$$S' =>_r^* \gamma B x =>_r \alpha\beta y,$$

для которых $FIRST_k(w) = FIRST_k(y)$ следует, что αAy = γBx.

То есть, если αβw и αβy – правовыводимые цепочки пополненной грамматики G', у которых $FIRST_k(w) = FIRST_k(y)$ и A→β – последнее правило, использованное в правом выводе цепочки αβw, то правило A→β должно использоваться также в правом разборе при свертке αβy к αAy.

Поскольку правило грамматики A→β не зависит от w, то из определения LR(k)-грамматики следует, что во множестве $FIRST_k(w)$ содержится информация достаточная для определения основы. **Основа – это** кодируемая цепочка символов в верхней части магазина.

Для любой LR(k)-грамматики G = (T, V, P, S) можно построить детерминированный анализатор, который выдает правый разбор входной цепочки. Анализатор состоит из магазина, входной ленты, выходной ленты и управляющего устройства (пара функций f и g) см. рис. 1.18.

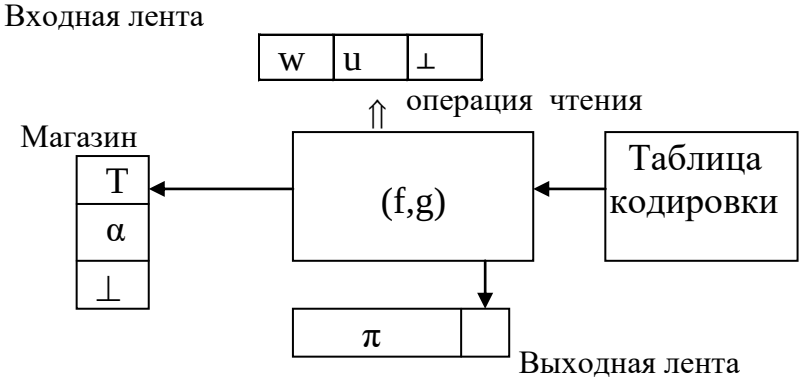


Рис. 1.18. LR(k)-анализатор

Магазинный алфавит V_p представляет собой множество специальных символов, соответствующих грамматическим вхождениям или их множествам.

Два способа построения LR(k) анализаторов

Существуют два способа построения LR(k) анализаторов:

1. Определение активных префиксов - использование расширенного магазинного алфавита (алгоритм перенос-свертка)
2. На основе грамматических вхождений.

Построим двумя способами LR(k) анализатор для заданной грамматики G = (T = {i, &, *,(,)}, V = {S, F, L}, P = { p_1: S→ F & L, p_2: S→ (S), p_3: F→ * L, p_4: F→ i, p_5: L→F }, S}

Построение LR(k) анализатора способом использования расширенного магазинного алфавита

Способ использования расширенного магазина состоит из трех шагов:

Шаг 1. Определение активных префиксов;

Шаг 2. Построение управляющей таблицы;

Шаг 3. Применение алгоритма «перенос-свёртка».

Шаг 1. Определение активных префиксов

Для каждого магазинного символа (за исключением S_0 и $\perp$), кодируемая цепочка является *префиксом* правой части некоторого правила грамматики. Например, магазинный символ «$\&_1$», находящийся на вершине магазина, означает что:

1. В верхушке магазина находится терминальный символ грамматики «&»
2. В верхней части магазина находится кодируемая этим символом цепочка символов «F&», то есть префикс *основы* – правой части правила 1.

Индекс каждого символа соответствует номеру правила, префикс правой части которого кодируется этим символом. И наоборот, каждый непустой префикс правой части правила кодируется некоторым магазинным символом. Например, правая часть правила 3 имеет два непустых префикса: «*» и «F», которые соответственно кодируются символами «$*_3$» и «L_3».

Символ переносится в магазин только в том случае, если он кодирует цепочку, «совместимую» с цепочкой, которая будет находиться в магазине после переноса.

Цепочка, кодируемая данным магазинным символом, *совместима* с цепочкой в магазине, если она является суффиксом магазинной цепочки после переноса данного символа.

Таблица 1. Закодированные символы для заданной грамматики.

Символ грамматики	Магазинный Символ	Кодируемая цепочка
S	S_0 S_2	$\perp$S (S
F	F_1 F_5	F F
L	L_1 L_3	F&L *L
I	i_4	I
(	$(_2$	(
*	$*_3$	*
&	$\&_1$	F&
)	$)_2$	(S)

Шаг 2. Построение управляющей таблицы;

Управление алгоритмом осуществляется при помощи двух функций, приведенных в таблице следующим образом:

1. Используя значение верхнего символа магазина и входного символа, алгоритм определяет значение функции действия: *ПЕРЕНОС или СВЕРТКА;*
2. При выполнении переноса определяется значение функции переходов, равное магазинному символу, который нужно втолкнуть в магазин;
3. Значение функции действия, равное СВЕРТКА(i), однозначно определяет этот шаг.

Работа алгоритма описывается в терминах конфигураций, представляющих собой тройки вида (αT, ax, π), где αT – цепочка магазинных символов (T-верхний символ магазина), ax – необработанная часть входной цепочки, π выход (строка из номеров правил), построенный к настоящему моменту времени.

Таблица состоит из двух подтаблиц – функции действия и функции переходов. Входным символам с ленты соответствуют столбцы таблицы, символам магазина – строки.

Алгоритм 3.11 Построение управляющей таблицы для LR(1) грамматики
Вход: LR(1)-грамматика G = (T, V, P, S)
Выход: Управляющая таблица M для грамматики G.

Функция действий f(u) определяется на множестве $(V_p \cup \{\perp\}) \times (T \cup \{\varepsilon\})$ по правилам:
1. Если A → β – правило грамматики с номером i, то для конфигурации (αT, ax, π), где T кодирует цепочку β, f(u) = C(i).
2. Если A → β – правило грамматики с номером i, то для конфигурации (αT, ax, π), где T кодирует некий префикс цепочки β (но не саму основу), f(u) = П.
3. Для конфигурации (S_0, ⊥, π), где S_0 кодирует цепочку ⊥S, f(u) = ДОПУСК.
4. В противном случае, f(u) = ОШИБКА.

Функция переходов g(X) определяется на множестве $(V_p \cup \{\perp\}) \times (V \cup T \cup \{\varepsilon\})$ по правилам:
1. Если для конфигурации (αT, ax, π) для входного символа a ϵ (V∪T) в таблице 1 существует символ a_i, совместимый с цепочкой αTa, то g(X) = a_i.
2. В противном случае, g(X) = ОШИБКА.

		функция действий f(u)					функция переходов g(X)							
	i	*	&	(	)	□	S	F	L	i	*	&	(	)
S_0	П	П	П	П	П	Д		F_1		i_4	$*_3$		$(_2$	
S_2	П	П	П	П	П			F_5		i_4	$*_3$		$(_2$	$)_2$
F_1	П	П	П	П	П							$\&_1$		
F_5	C(5)	C(5)	C(5)	C(5)	C(5)	C(5)								
L_1	C(1)	C(1)	C(1)	C(1)	C(1)	C(1)								
L_3	C(3)	C(3)	C(3)	C(3)	C(3)	C(3)								
i_4	C(4)	C(4)	C(4)	C(4)	C(4)	C(4)								

$(_2$	П	П	П	П	П		S_2	F_1		i_4	$*_3$		$(_2$	
$*_3$	П	П	П	П	П			F_5	L_3	i_4	$*_3$			
$\&_1$	П	П	П	П	П			F_5	L_1	i_4	$*_3$			
$)_2$	C(2)	C(2)	C(2)	C(2)	C(2)	C(2)								
□	П	П	П	П	П		S_0	F_1		i_4	$*_3$		$(_2$	

Шаг 3. Применение алгоритма «перенос-свёртка».

Алгоритм 3.12. LR(k)-алгоритм разбора

Вход: Анализируемая цепочка $z = t_1t_2\ldots\mathbf{t_j}\ldots t_n \in T^*$, где j - номер текущего символа входной цепочки, находящегося под читающей головкой. Управляющая таблица M (множество LR(k)-таблиц) для LR(k)-грамматики G= (T, V, P, S)

Выход: Если $z \in L(G)$, то правый разбор цепочки z, в противном случае – сигнал об ошибке.

Алгоритм.

1. j := 0.

2. j := j+1. Если j > n, то выдать сообщение об ошибке и перейти к шагу 5.

3. Определить цепочку u следующим образом:

- если k = 0, то u= $\mathbf{t_j}$;
- если $k \geq 1$ и $j + k - 1 \leq n$, то u= $t_j\mathbf{t_{j+1}\ldots t_{j+k-1}}$ – первые k-символов цепочки $t_jt_{j+1}\ldots t_n$;
- если $k \geq 1$ и $j + k - 1 > n$, то u= $t_j\mathbf{t_{j+1}\ldots t_n}$ – остаток входной цепочки.

4. Применить функцию действия f из строки таблицы M, отмеченной верхним символом магазина T, к цепочке u:

f(u) = ПЕРЕНОС (П). Определить функцию перехода $g(t_j)$ из строки таблицы M, отмеченной символом T из верхушки магазина. Если $g(t_j) = T'$ и $T' \in V_p \cup \{\perp\}$, то записать T' в магазин и перейти к шагу 2. Если $g(t_j)$= ОШИБКА, то выдать сигнал об ошибке и перейти к шагу 5;

- f(u) = (СВЕРТКА, i) (С) и $A \rightarrow \alpha$ – правило вывода с номером i грамматики G. Удалить из верхней части магазина $|\alpha|$ символов, в результате чего в верхушке магазина окажется символ $T' \in V_p \cup \{\perp\}$, и выдать номер правила i на входную ленту. Определить символ T''= g(A) из строки таблицы M, отмеченной символом T', записать его в магазин и перейти к шагу 3.
- f(u) = ОШИБКА (О). Выдать сообщение об ошибке и перейти к шагу 5.
- f(u) = ДОПУСК (Д). Объявить цепочку, записанную на входной ленте, правым разбором входной цепочки z.

5. Останов.

Пример

Рассмотрим, например, строку таблицы, отмеченную закодированным символом $\&_1$, который представляет собой префикс правой части правила p_1: $S \rightarrow F \,\&\, L$. Пусть i – текущий входной символ. Согласно правилу p_1 за $\&_1$ может следовать нетерминал L_1 или символ, представляющий подцепочку, порождаемую L_1. Цепочка, порождаемая L_1, может начинаться либо с i_4

(правило p_4), либо с F_5 (правило p_5), которое в свою очередь порождает цепочку, начинающуюся с $*_3$ или снова i_4. Следовательно, для входного символа i в магазин можно втолкнуть только символ i_4, который является кодированным представлением префикса правой части правила p_4.

Строка таблицы, отмеченная маркером дна магазина (⊥), соответствует начальной конфигурации алгоритма. В первый момент времени в магазин можно записать символ S_0 или символы F_1, i_4, $*_1$, ($_1$, которые представляют собой префиксы цепочек, выводимых из начального вхождения S_0 (грамматического вхождения начального символа грамматики S, входящего в правую часть нулевого правила вывода пополненной грамматики).

Рассмотрим последовательность тактов LR(k)-алгоритма при анализе входной цепочки (*i&i):

Шаг 1. Начальная конфигурация алгоритма – (⊥, (*i&i)⊥, ε) (в магазине находится маркер дна магазина, текущий входной символ – «(»).

Для строки управляющей таблицы, отмеченной символом ⊥, f(( ) = ПЕРЕНОС и g(( ) = ($_2$, поэтому:

- В магазин записывается символ ($_2$.
- Входная головка сдвигается на один символ вправо.

Алгоритм переходит в конфигурацию (⊥($_2$,*i&i)⊥,ε).

Шаг 2. Для строки таблицы, отмеченной символом ($_2$, f(*) = ПЕРЕНОС и g(*) = $*_3$, поэтому алгоритм перейдёт в конфигурацию (⊥($_2*_3$,i&i)⊥,ε). Аналогично для магазинного символа $*_3$ и следующего входного символа i, алгоритм перейдёт в конфигурацию (⊥($_2*_3i_4$, &i)⊥, ε).

Шаг 3. Для строки таблицы, отмеченной символом i_4, f(&) = (C, 4), значит, необходимо выполнить свёртку по правилу p_4: F→i.

Правая часть этого правила содержит только один символ, поэтому:

- Удаляем из магазина символ i_4;
- Определяем значение функции переходов для символа F из левой части правила p_4 в строке таблицы M, отмеченной символом $*_3$, который теперь стал верхним символом магазина. Значение функции переходов: g(F) = F_5.
- Записываем в выходную цепочку число 4 (номер примененного правила).

Алгоритм переходит в конфигурацию (⊥($_2*_3F_5$, &i)⊥,4).

Продолжая, получим следующую последовательность тактов:

(⊥($_2*_3F_5$, &i)⊥, 4) ⊢ (⊥($_2*_3L_3$, &i)⊥, 45) ⊢ (⊥($_2F_1$, &i)⊥, 453) ⊢ (⊥($_2F_1$&$_1$, i)⊥, 453) ⊢ (⊥($_2F_1$&$_1i_4$,)⊥, 453) ⊢ (⊥($_2F_1$&$_1F_5$,)⊥, 4534) ⊢ (⊥($_2F_1$&$_1L_1$,)⊥, 45345) ⊢ (⊥($_2S_2$,)⊥, 453451) ⊢ (⊥($_2S_2$)$_2$, ⊥, 453451) ⊢ (⊥S_2, ⊥, 4534512) ⊢ ДОПУСК.

Пусть анализируется цепочка (*i&&i), содержащая синтаксическую ошибку, последовательность тактов будет следующая:

(⊥, (*i&&i)⊥, ε) ⊢ (⊥($_2$,*i&&i)⊥,ε) ⊢ (⊥($_2*_3$,i&i)⊥,ε) ⊢ (⊥($_2*_3i_4$, &i)⊥, ε) ⊢ (⊥($_2*_3F_5$, &i)⊥,4) ⊢ (⊥($_2*_3L_3$, &i)⊥, 45) ⊢ (⊥($_2F_1$, &i)⊥, 453) ⊢ (⊥($_2F_1$&$_1$, i)⊥, 453) ⊢ ОШИБКА.

Для строки таблицы, отмеченной символом $\&_1$ f(&) = ПЕРЕНОС, но значение функции g(&) = ОШИБКА. Алгоритм должен выдать сообщение об ошибке и завершить работу.

Построение LR(k) анализатора способом грамматических вхождений

Способ использования грамматических вхождений состоит из четырёх шагов:
Шаг 1. Определение грамматических вхождений (см. алгоритм фреймворка);
Шаг 2. Построение конечного автомата из грамматических вхождений;
Шаг 3. Построение управляющей таблицы.
Шаг 4. Применение алгоритма «перенос-свёртка».

Шаг 1. Определение грамматических вхождений
Грамматическое вхождение – это символы полного словаря грамматики, снабженные двумя индексами. Первый индекс i задает номер правила грамматики, в правую часть которого входит данный символ, а второй индекс j – номер позиции символа в этой правой части.

Пример
В приведённой выше грамматике с правилами:
p_1: S→ F & L,
p_2: S→ (S),
p3: F→ * L,
p_4: F→ i,
p_5: L→F
можно, например, выделить грамматические вхождения $\&_{1,2}$, $)_{2,3}$, $L_{3,2}$ и т.д.
Существуют несколько способов построения детерминированного конечного автомата, распознающего активные префиксы [2, 3, 28, 37]. Рассмотрим вначале достаточно простой способ построения канонической системы LR(0)-ситуаций, приведенный в [2], для чего определим две функции: CLOSURE и GOTO.

Шаг 2. Построение конечного автомата из грамматических вхождений

LR(0) ситуацией назовём запись $[A \rightarrow w_1 \bullet w_2]$, если $A \rightarrow w_1 w_2$ – правило КС-грамматики.

Пример.
Так как LR(0) - ситуация - это просто правило грамматики с точкой в некоторой позиции правой части, то для правила S → (S) можно получить 4 ситуации:
[S → • (S)]
[S → (•S)]
[S → (S•)]
[S → (S)•]

Замечание. LR(0)-ситуация не содержит аванцепочку u, поэтому при ее записи можно опускать квадратные скобки.

Для каждого активного префикса существует непустое множество ситуаций. Основная идея построения простого LR(0)-анализатора состоит в том, чтобы вначале построить на базе КС-грамматики детерминированный конечный автомат для распознавания активных префиксов. Для этого ситуации группируются в множества, которые приводят к состояниям анализатора.

Пусть I – множество LR(0)-ситуаций КС-грамматики G. Тогда назовем замыканием множества I множество ситуаций CLOSURE(I), построенное по следующим правилам:

1. Включить в CLOSURE(I) все ситуации из I.

2. Если ситуация A → α •Bβ уже включена в CLOSURE(I) и B → γ - правило грамматики, то добавить в множество CLOSURE(I) ситуацию B → • γ при условии, что там ее еще нет.

3. Повторять правило 2, до тех пор, пока в CLOSURE(I) нельзя будет включить новую ситуацию.

Второе правило построения можно пояснить следующим образом:

- Наличие ситуации A → α •Bβ в множестве CLOSURE(I) говорит о том, что в некоторый момент разбора мы можем встретить в входном потоке анализатора подстроку, выводимую из Bβ.
- Если в грамматике имеется правило B → γ, то мы также можем встретить во входном потоке анализатора подстроку, выводимую из γ, следовательно, в CLOSURE(I) нужно включить ситуацию B → • γ.

Рассмотрим пополненную грамматику G_0, содержащую следующие правила:

p_0: S' → S
p_1: S → F & L,
p_2: S → (S),
p_3: F → * L,
p_4: F → i,
p_5: L →F

Пусть вначале множество ситуации CLOSURE(I) содержит одну ситуацию *S′* → • *S*, т.е. CLOSURE(I) ={ *S′* → • *S* }

Ситуация S'→ •S содержит точку перед символом S, поэтому по второму правилу в CLOSURE(*I*) необходимо включить *S*-правила с точкой слева. Окончательно получим:

CLOSURE(*I*) ={*S′* → •*S, S* → •*F&L, S* → • *(S), F* → •L, F* → •*i* }.**

Если *I*-множество ситуаций, допустимых для некоторого активного префикса γ , то GOTO(*I,X*) – это множество ситуаций, допустимых для активного префикса γ*X*.

Аргументами функции GOTO(*I,X*) являются множество ситуаций *I* и символ грамматики *X*.

Определение. Функция GOTO(*I,X*) определяется как замыкание множества всех ситуаций [$A \rightarrow \alpha X \cdot \beta$], таких что [$A \rightarrow \alpha \cdot X\beta$] ϵ *I*

Пусть для грамматики G_0 некоторое множество *I = {[S′ → S•], [S → F•&L]}.*

Для вычисления значения GOTO(I, &) необходимо рассмотреть ситуации, в которых непосредственно сразу за точкой следует символ &. Это ситуация *[S → F•&L]*. Перемещая точку за символ &, построим множество ситуаций *[S → F&•L]* и замыкание этого множества: *{L→ •F, F→ •i, F→•*L}.* Тогда GOTO(I, &) = *{[S → F&•L], [L→ •F], [F→ •i], [F→•*L]}.*

В [2] и [3] приведены алгоритмы, позволяющие построить каноническую систему множеств *LR(0)* –ситуаций. Процесс построения канонической системы множеств *LR(0)* –ситуаций можно описать с помощью следующих действий:

1. $\varphi = \emptyset$
2. Включить в φ множество A_0 = *CLOSURE([S'→ • S])*, которое в начале «не отмечено».
3. Если множество ситуаций *A*, входящее в систему , «не отмечено», то:
   - отметить множество *A*;
   - вычислить для каждого символа $X \epsilon (V \cup \Sigma)$ значение $A' = GOTO(A,X)$;
   - если множество $A' \neq \emptyset$ и еще не включено в φ, то включить его в систему множеств как «неотмеченное» множество.
4. Повторять шаг 3, пока все множества ситуаций системы φ не будут отмечены.

Пример

Построим систему множеств φ для нашей пополненной граматики G_0:

В начале построения система множеств $\varphi = \emptyset$.

Определив множество $A_0 = CLOSURE(\{|S' \rightarrow \bullet\ S|\}) = \{S' \rightarrow \bullet\ S, S \rightarrow \bullet\ F\&L, S \rightarrow \bullet\ (S), F \rightarrow \bullet\ i, F \rightarrow \bullet * L\}$, включим его в систему φ в качестве «неотмеченного» множества.

A_0	$S' \rightarrow \bullet\ S$ $S \rightarrow \bullet\ F\&L$ $S \rightarrow \bullet\ (S)$ $F \rightarrow \bullet\ i$ $F \rightarrow \bullet * L$

Теперь мы должны отметить множество A_0 и определить множества *GOTO(A_0, X)* для всех символов X грамматики G_0: $A_1 = GOTO(A_0, S) = CLOSURE(\{S' \rightarrow S\ \bullet\}) = \{S' \rightarrow S\ \bullet\}$

Множество A_1 непустое и отсутствует в системе φ. Включим A_1 в φ как «неотмеченное» множество:

A_0	$S' \rightarrow \bullet\ S$ $S \rightarrow \bullet\ F\&L$

	$S \to\bullet\ (S)$ $F \to\bullet\ i$ $F \to\bullet *\ L$
A_1	$S' \to S\ \bullet$

Продолжая строить *GOTO(A0, X)*, получаем:

$A_2 = GOTO(A_0, (\,) = CLOSURE([S \to (\bullet\ S)]) = \{S \to (\bullet\ S), S \to\bullet\ (S), S \to\bullet\ F\&L,\ F \to\bullet\ i, F \to\bullet *\ L\ \} A_3 = GOTO(A_0, i) = \{F \to i\ \bullet\}$

$A_4 = GOTO(A_0, *) = \{F \to *\bullet\ L, L \to\bullet\ \mathrm{F}, F \to\bullet *\ \mathrm{L}, F \to\bullet\ \mathrm{i}\}$

$A_5 = GOTO(A_0, F) = \{S \to F\ \bullet\ \&L\}$

Остальные множества *GOTO(A0, X)*, где $X \in \{L, \&,)\}$, пусты, поэтому система φ принимает вид:

A_0	$S' \to\bullet\ S$ $S \to\bullet\ F\&L$ $S \to\bullet\ (S)$ $F \to\bullet\ i$ $F \to\bullet *\ L$
A_1	$S' \to S\ \bullet$
A_2	$S \to (\bullet\ S)$ $S \to\bullet\ (S)$ $S \to\bullet\ F\&L$ $F \to\bullet\ i$ $F \to\bullet *\ \mathrm{L}$
A_3	$F \to i\ \bullet$
A_4	$F \to *\bullet\ L$ $L \to\bullet\ \mathrm{F}$ $F \to\bullet *\ \mathrm{L}$ $F \to\bullet\ \mathrm{i}$
A_5	$S \to F\ \bullet\ \&L$

Продолжая выполнять шаг 3 построения системы множеств, «отмечаем» множество A_1. Для всех $X \in (V \cup \Sigma)$ множества *GOTO(A1, X)* пусты.

Теперь отмечаем множество A_2. $A_6 = GOTO(A_2, S) = \{S \to (S\ \bullet)\}$

Добавляем множество A_6 в φ . $A_7 = GOTO(A_2, (\,) = CLOSURE\big(S \to (\bullet\ S)\big) = \{S \to (\bullet\ S), S \to\bullet\ (S), S \to\bullet\ F\&L,\ F \to\bullet\ i, F \to\bullet *\ \mathrm{L}\}$

Такое множество (само множество A_2) уже есть в φ.

Множества *GOTO(A2, F), GOTO(A2, i), GOTO(A2, *)* также уже включены в φ. Для остальных символов $X \in (V \cup \Sigma)$ множества *GOTO(A2, X)* пусты.

Отмечаем множество A_3. Для всех $X \in (V \cup \Sigma)$ множества *GOTO(A3, X)* пусты.

Продолжая аналогичным образом, включаем в систему φ множества:

$A_7 = GOTO(A_4, L) = \{F \to *\ \mathrm{L}\ \bullet)\}$

$A_8 = GOTO(A_4, F) = \{L \to F\ \bullet)\}$

$A_9 = GOTO(A_5, \&) = \{S \rightarrow \text{F\&} \bullet L, L \rightarrow\bullet \text{F}, F \rightarrow\bullet \text{i}, F \rightarrow\bullet* \text{L}\}$

$A_{10} = GOTO(A_6,)) = \{S \rightarrow (S) \bullet\}$

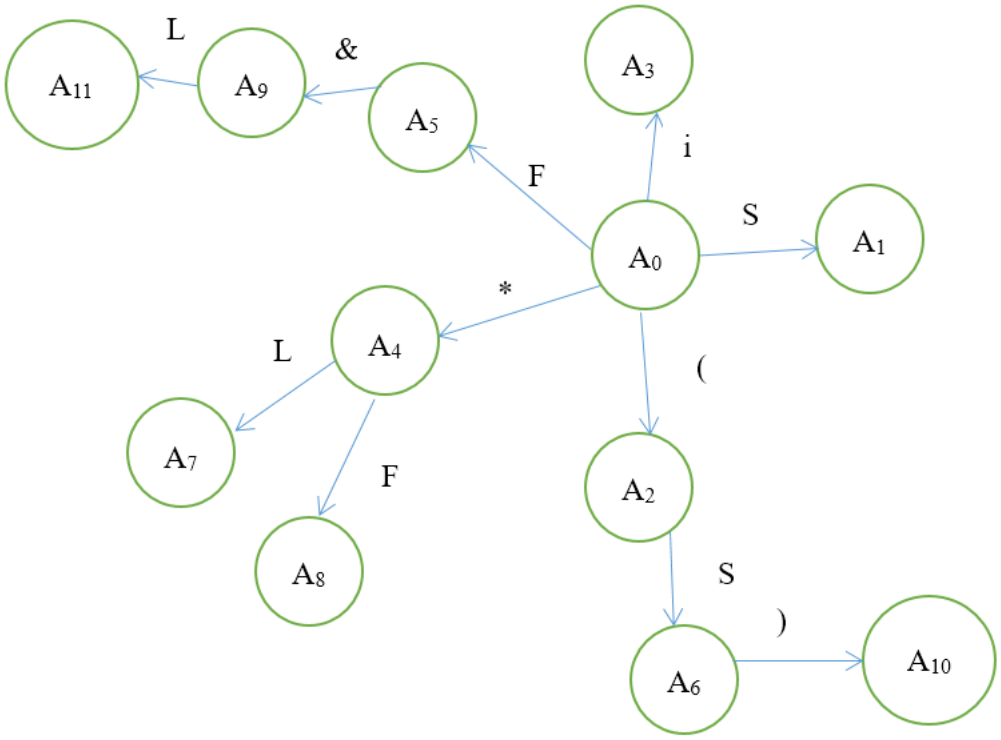
$A_{11} = GOTO(A_9, L) = \{S \rightarrow \text{F\&}L \bullet\}$

В результате получаем окончательную систему LR(0) – множеств φ:

A_0	$S' \rightarrow\bullet S$ $S \rightarrow\bullet F\&L$ $S \rightarrow\bullet (S)$ $F \rightarrow\bullet i$ $F \rightarrow\bullet* L$
A_1	$S' \rightarrow S \bullet$
A_2	$S \rightarrow (\bullet S)$ $S \rightarrow\bullet (S)$ $S \rightarrow\bullet F\&L$ $F \rightarrow\bullet i$ $F \rightarrow\bullet*$ L
A_3	$F \rightarrow i \bullet$
A_4	$F \rightarrow*\bullet L$ $L \rightarrow\bullet$ F $F \rightarrow\bullet*$ L $F \rightarrow\bullet$ i
A_5	$S \rightarrow F \bullet \&L$
A_6	$S \rightarrow (S \bullet)$
A_7	$F \rightarrow*$ L $\bullet$
A_8	$L \rightarrow F \bullet$
A_9	$S \rightarrow$ F& $\bullet L$ $L \rightarrow\bullet$ F $F \rightarrow\bullet$ i $F \rightarrow\bullet*$ L
A_{10}	$S \rightarrow (S) \bullet$
A_{11}	$S \rightarrow$ F&$L \bullet$

Используя каноническую систему LR(0) – множеств, можно представить функцию GOTO в виде диаграммы детерминированного конечного автомата.

Диаграмма переходов ДКА для активных префиксов грамматики G0:



Переход в состояние, которому на диаграмме соответствует лист (вершина, не имеющая исходящих дуг) однозначно определяет функцию свёртки с возвратом к предыдущему состоянию.

Шаг 3. Построение управляющей таблицы.

Алгоритм построения управляющей таблицы M для LR(0)-грамматик основывается на рассмотрении пар грамматических вхождений, которые могут быть представлены соседними магазинными символами в процессе разбора допустимых цепочек.

Управляющая таблица:

	(	)	&	I	*	$	S	F	L
0	П(2)			П(3)	П(4)			5	
1						доп уск			
2							6		
3	С(4)	С(4)	С(4)	С(4)	С(4)	С(4)			
4								8	7
5			П(9)						
6		П(10)							
7	С(3)	С(3)	С(3)	С(3)	С(3)	С(3)			
8	С(5)	С(5)	С(5)	С(5)	С(5)	С(5)			
9			П(11)						

10	C(2)	C(2)	C(2)	C(2)	C(2)				
11	C(1)	C(1)	C(1)	C(1)	C(1)				

Шаг 4. Применение алгоритма «перенос-свёртка».

Работа алгоритма описывается в терминах конфигураций, представляющих собой тройки вида (αT, ax, π), где αT – цепочка магазинных символов (T-верхний символ магазина), ax – необработанная часть входной цепочки, π – выход, построенный к настоящему моменту времени.

Рассмотрим послендовательность тактов LR(k)-алгоритма при анализе входной цепочки abcb. В магазине МП-автомата вместе с помещенным туда символами показаны и номера строк управляющей таблицы, в формате «символ»«номер».

Пример

Рассмотрим последовательность тактов алгоритма при анализе входной цепочки i&**i

(⊥0, i&**i$, ε)⊢(⊥0i3, &**i$, ε) ⊢(⊥0F5, &**i$, 4) ⊢(⊥0F5&9, **i$, 4) ⊢(⊥0F5&9*4, *i$, 4) ⊢ (⊥0F5&9*4*4, i$, 4) ⊢ (⊥0F5&9*4*4i3, $, 4) ⊢(⊥0F5&9*4*4F8, $, 44)⊢ (⊥0F5&9*4*4L7, $, 445) ⊢ (⊥0F5&9*4F8, $, 4453) ⊢ (⊥0F5&9*4L7, $, 44535) ⊢ (⊥0F5&9F8, $, 445353) ⊢ (⊥0F5&9L11, $, 4453535)⊢(⊥0S1, $, 44535351) ⊢ ДОПУСК

3.8. Грамматики предшествования

Существует подкласс LR(k)-грамматик, который называется грамматиками предшествования для них легко строится алгоритм типа “перенос-свертка”. При восходящих методах синтаксического анализа в текущей сентенциальной форме выполняется поиск основы α, которая в соответствии с правилом грамматики A→ α сворачивается к нетерминальному символу A. Основная проблема восходящего синтаксического анализа заключается в поиске основы и нетерминального символа, к которому ее нужно приводить (сворачивать).

Принцип организации распознавателя входных цепочек языка, заданного грамматикой предшествования, основывается на том, что для каждой упорядоченной пары символов в грамматике устанавливается некоторое отношение, называемое отношением предшествования. В процессе разбора входной цепочки расширенный МП-автомат сравнивает текущий символ входной цепочки с одним из символов, находящихся на верхушке стека автомата. В процессе сравнения проверяется, какое из возможных отношений предшествования существует между этими двумя символами. В зависимости от найденного отношения выполняется либо сдвиг (перенос), либо свертка. При отсутствии отношения предшествования между символами алгоритм сигнализирует об ошибке.

Задача заключается в том, чтобы иметь возможность непротиворечивым образом определить отношения предшествования между символами грамматики. Если это возможно, то грамматика может быть отнесена к одному из классов грамматик предшествования.

Существует несколько видов грамматик предшествования. Они различаются по тому, какие отношения предшествования в них определены и между какими типами символов (терминальными или нетерминальными) могут быть установлены эти отношения. Кроме того, возможны незначительные модификации функционирования самого алгоритма «сдвиг-свертка» в распознавателях для таких грамматик (в основном на этапе выбора правила для выполнения свертки, когда возможны неоднозначности) [5, 6, 23, 65].

Выделяют следующие виды грамматик предшествования:

- простого предшествования;
- расширенного предшествования;
- слабого предшествования;
- смешанной стратегии предшествования;
- операторного предшествования.

Рассмотрим два наиболее простых и распространенных типа — грамматики простого и операторного предшествования.

Грамматики простого предшествования

Грамматикой простого предшествования называют такую приведенную (без циклов, бесплодных и недостижимых символов и λ-правил)КС-грамматику G(**VN,VT,P**,S), **V** = **VT**∪**VN**, в которой:

1. Для каждой упорядоченной пары терминальных и нетерминальных символов выполняется не более чем одно из трех отношений предшествования:

- $B_i \doteq B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если и только если $\exists$ правило $A \to xB_iB_jy \in \mathbf{P}$, где $A \in \mathbf{VN}$, $x,y \in \mathbf{V^*}$;
- $B_i \lessdot B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если и только если $\exists$ правило $A \to xB_iDy \in \mathbf{P}$ и вывод $D \Rightarrow^* S_jz$, где $A,D \in \mathbf{VN}$, $x,y,z \in \mathbf{V^*}$;
- $B_i \gtrdot B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если и только если $\exists$ правило $A \to xCB_jy \in \mathbf{P}$ и вывод $C \Rightarrow^* zB_i$ или $\exists$ правило $A \to xCDy \in \mathbf{P}$ **и** выводы $C \Rightarrow^* zB_i$ и $D \Rightarrow^* B_jw$, где $A,C,D \in \mathbf{VN}$, $x,y,z,w \in \mathbf{V^*}$.

2. Различные правила в грамматике имеют разные правые части (то есть в грамматике не должно быть двух различных правил с одной и той же правой частью).

Отношения $\doteq$, $\lessdot$ и $\gtrdot$ называют отношениями предшествования для символов. Отношение предшествования единственно для каждой упорядоченной пары символов. При этом между какими-либо двумя символами может и не быть отношения предшествования — это значит, что они не могут находиться рядом ни в одном элементе разбора синтаксически правильной цепочки. Отношения предшествования зависят от порядка, в котором стоят символы, и в этом смысле их нельзя путать со знаками математических операций — они не обладают ни свойством коммутативности, ни свойством ассоциативности. Например, если известно, что $B_i \gtrdot B_j$, то не обязательно выполняется $B_j \lessdot B_i$ (поэтому знаки предшествования иногда помечают специальной точкой: $\doteq$, $\lessdot$, $\gtrdot$).

Для грамматик простого предшествования известны следующие полезные свойства:

- всякая грамматика простого предшествования является однозначной;
- легко проверить, является или нет произвольная КС-грамматика грамматикой простого предшествования (для этого достаточно проверить рассмотренные выше свойства грамматик простого предшествования или воспользоваться алгоритмом построения матрицы предшествования, который рассмотрен далее).

Как и для многих других классов грамматик, для грамматик простого предшествования не существует алгоритма, который бы мог преобразовать произвольную КС-грамматику в грамматику простого предшествования (или доказать, что преобразование невозможно).

Метод предшествования основан на том факте, что отношения предшествования между двумя соседними символами распознаваемой строки соответствуют трем следующим вариантам:

- $B_i \lessdot B_{i+1}$, если символ B_{i+1} — крайний левый символ некоторой основы (это отношение между символами можно назвать «предшествует основе» или просто «предшествует»);
- $B_i \gtrdot B_{i+1}$, если символ B_i — крайний правый символ некоторой основы (это отношение между символами можно назвать «следует за основой» или просто «следует»);

- $B_i =\bullet B_{i+1}$, если символы B_i и B_{i+1} принадлежат одной основе (это отношение между символами можно назвать «составляют основу»),

Исходя из этих соотношений, выполняется разбор строки для грамматики предшествования.

Суть принципа такого разбора можно пояснить на рис. 12.5. На нем изображена входная цепочка символов $\alpha\beta\gamma\delta$ в тот момент, когда выполняется свертка цепочки γ. Символ *a* является последним символом подцепочки α, а символ *b* — первым символом подцепочки β. Тогда, если в грамматике удастся установить непротиворечивые отношения предшествования, то в процессе выполнения разбора по алгоритму «сдвиг-свертка» можно всегда выполнять сдвиг до тех пор, пока между символом на верхушке стека и текущим символом входной цепочки существует отношение **<•** или **=•**. А как только между этими символами будет обнаружено отношение **•>**, так сразу надо выполнять свертку. Причем для выполнения свертки из стека надо выбирать все символы, связанные отношением **=•**. То, что все различные правила в грамматике предшествования имеют различные правые части, гарантирует непротиворечивость выбора правила при выполнении свертки.

Таким образом, установление непротиворечивых отношений предшествования между символами грамматики в комплексе с несовпадающими правыми частями различных правил дает ответы на все вопросы, которые надо решить для организации работы алгоритма «сдвиг-свертка» без возвратов.

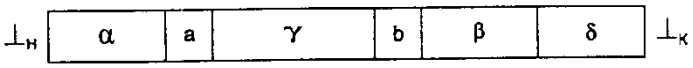


Рис. 12.5. Отношения между символами входной цепочки в грамматике предшествования

На основании отношений предшествования строят матрицу предшествования грамматики. Строки матрицы предшествования помечаются первыми (левыми) символами, столбцы — вторыми (правыми) символами отношений предшествования. В клетки матрицы на пересечении соответствующих столбца и строки помещаются знаки отношений. При этом пустые клетки матрицы говорят о том, что между данными символами нет ни одного отношения предшествования.

Матрицу предшествования грамматики сложно построить, опираясь непосредственно на определения отношений предшествования. Удобнее воспользоваться двумя дополнительными множествами — множеством крайних левых и множеством крайних правых символов относительно нетерминальных символов грамматики G(**VN,VT,P**,S), **V** = **VT**∪**VN**. Эти множества определяются следующим образом:

- **L**(A) = {X | ∃ A⇒*Xz}, A∈**VN**, X∈**V**, z∈**V*** — множество крайних левых символов относительно нетерминального символа A (цепочка z может быть и пустой цепочкой);
- R(A) = {X | ∃ A⇒*zX}, A∈**VN**, X∈**V**, z∈**V*** — множество крайних правых символов относительно нетерминального символа A.

Иными словами, множество крайних левых символов относительно нетерминального символа A — это множество всех крайних левых символов в цепочках, которые могут быть выведены из символа A. Аналогично, множество крайних правых символов относительно нетерминального символа A — это множество всех крайних правых символов в цепочках, которые могут быть выведены из символа A. Тогда отношения предшествования можно определить так:

- $B_i =\bullet B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если $\exists$ правило $A \to xB_iB_jy \in \mathbf{P}$, где $A \in \mathbf{VN}$, $x,y \in \mathbf{V}^*$;
- $B_i <\bullet B_j$ ($\forall B_i, B_j \in \mathbf{V}$), если $\exists$ правило $A \to xB_iDy \in \mathbf{P}$ и $B_j \in \mathbf{L}(D)$, где $A,D \in \mathbf{VN}$, $x,y \in \mathbf{V}^*$;
- $B_i \bullet> Bj$ ($\forall B_i, B_j \in \mathbf{V}$), если $\exists$ правило $A \to xCB_jy \in \mathbf{P}$ и $B_i \in \mathbf{R}(C)$ или $\exists$ правило $A \to xCDy \in \mathbf{P}$ и $B_i \in \mathbf{R}(C)$, $B_j \in \mathbf{L}(D)$, где $A,C,D \in \mathbf{VN}$, $x,y \in \mathbf{V}^*$.

Такое определение отношений удобнее на практике, так как не требует построения выводов, а множества $\mathbf{L}(A)$ и $\mathbf{R}(A)$ могут быть построены для каждого нетерминального символа $A \in \mathbf{VN}$ грамматики $G(\mathbf{VN,VT,P},S)$, $\mathbf{V} = \mathbf{VT} \cup \mathbf{VN}$ по очень простому алгоритму.

Шаг 1. $\forall A \in \mathbf{VN}$:

$\mathbf{R}_o(A) = \{X \mid A \to yX, X \in \mathbf{V}, y \in \mathbf{V}^*\}$, $\mathbf{L}_0(A) = \{X \mid A \to Xy, X \in \mathbf{V}, y \in \mathbf{V}^*\}$, i := 1.

Для каждого нетерминального символа A ищем все правила, содержащие A в левой части. Во множество $\mathbf{L}(A)$ включаем самый левый символ из правой части правил, а во множество $\mathbf{R}(A)$ — самый крайний правый символ из правой части. Переходим к шагу 2.

Шаг 2. $\forall A \in \mathbf{VN}$:

$\mathbf{R}_i(A) = \mathbf{R}_{i-1}(A) \cup \mathbf{R}_{i-1}(B)$, $\forall B \in (\mathbf{R}_{i-1}(A) \cap \mathbf{VN})$,

$\mathbf{L}_i(A) = \mathbf{L}_{i-1}(A) \cup \mathbf{L}_{i-1}(B)$, $\forall B \in (\mathbf{L}_{i-1}(A) \cap \mathbf{VN})$.

Для каждого нетерминального символа A: если множество $\mathbf{L}(A)$ содержит нетерминальные символы грамматики A', A", …, то его надо дополнить символами, входящими в соответствующие множества $\mathbf{L}(A')$, $\mathbf{L}(A")$, ... и не входящими в $\mathbf{L}(A)$. Ту же операцию надо выполнить для $\mathbf{R}(A)$.

Шаг 3. Если $\exists A \in \mathbf{VN}$: $\mathbf{R}_i(A) \neq \mathbf{R}_{i-1}(A)$ или $\mathbf{L}_i(A) \neq \mathbf{L}_{i-1}(A)$, то i:=i+l и вернуться к шагу 2, иначе построение закончено: $\mathbf{R}(A) = \mathbf{R}_i(A)$ и $\mathbf{L}(A) = \mathbf{L}_i(A)$.

Если на предыдущем шаге хотя бы одно множество $\mathbf{L}(A)$ или $\mathbf{R}(A)$ для некоторого символа грамматики изменилось, то надо вернуться к шагу 2, иначе построение закончено.

После построения множеств $\mathbf{L}(A)$ и $\mathbf{R}(A)$ по правилам грамматики создается матрица предшествования. Матрицу предшествования дополняют символами $\perp_H$ и $\perp_K$ (начало и конец цепочки). Для них определены следующие отношения предшествования:

$\perp_H <\bullet X$, $\forall a \in \mathbf{V}$, если $\exists S \Rightarrow^* Xy$, где $S \in \mathbf{VN}$, $y \in \mathbf{V}^*$ или (с другой стороны) если $X \in \mathbf{L}(S)$;

$\perp_K \bullet> X$, $\forall a \in \mathbf{V}$, если $\exists S \Rightarrow^* yX$, где $S \in \mathbf{VN}$, $y \in \mathbf{V}^*$ или (с другой стороны) если $X \in \mathbf{R}(S)$;

Здесь S — целевой символ грамматики.

Матрица предшествования служит основой для работы распознавателя языка, заданного грамматикой простого предшествования.

Алгоритм «сдвиг-свертка» для грамматики простого предшествования

Данный алгоритм выполняется расширенным МП-автоматом с одним состоянием. Отношения предшествования служат для того, чтобы определить в процессе выполнения алгоритма, какое действие — сдвиг или свертка — должно выполняться на каждом шаге алгоритма, и однозначно выбрать цепочку для свертки. Однозначный выбор правила при свертке обеспечивается за счет различия правых частей всех правил грамматики. В начальном состоянии автомата считывающая головка обозревает первый символ входной цепочки, в стеке МП-автомата находится символ $\perp_{\text{Н}}$ (начало цепочки), в конец цепочки помещен символ $\perp_{\text{К}}$ (конец цепочки). Символы $\perp_{\text{Н}}$ и $\perp_{\text{К}}$ введены для удобства работы алгоритма, в язык, заданный исходной грамматикой, они не входят.

Разбор считается законченным (алгоритм завершается), если считывающая головка автомата обозревает символ $\perp_{\text{К}}$ и при этом больше не может быть выполнена свертка. Решение о принятии цепочки зависит от содержимого стека. Автомат принимает цепочку, если в результате завершения алгоритма в стеке находятся начальный символ грамматики S и символ $\perp_{\text{Н}}$. Выполнение алгоритма может быть прервано, если на одном из его шагов возникнет ошибка. Тогда входная цепочка не принимается. Алгоритм состоит из следующих шагов.

Шаг 1. Поместить в верхушку стека символ $\perp_{\text{Н}}$, считывающую головку — в начало входной цепочки символов.

Шаг 2. Сравнить с помощью отношения предшествования символ, находящийся на вершине стека (левый символ отношения), с текущим символом входной цепочки, обозреваемым считывающей головкой (правый символ отношения).

Шаг 3. Если имеет место отношение **<•** или **=•**, то произвести сдвиг (перенос текущего символа из входной цепочки в стек и сдвиг считывающей головки на один шаг вправо) и вернуться к шагу 2. Иначе перейти к шагу 4.

Шаг 4. Если имеет место отношение **•>**, то произвести свертку. Для этого надо найти на вершине стека все символы, связанные отношением **=•** («основу»), удалить эти символы из стека. Затем выбрать из грамматики правило, имеющее правую часть, совпадающую с основой, и поместить в стек левую часть выбранного правила (если символов, связанных отношением =•<$]command>, на верхушке стека нет, то в качестве основы используется один, самый верхний символ стека). Если правило, совпадающее с основой, найти не удалось, то необходимо прервать выполнение алгоритма и сообщить об ошибке, иначе, если разбор не закончен, то вернуться к шагу 2.

Шаг 5. Если не установлено ни одно отношение предшествования между текущим символом входной цепочки и символом на верхушке стека, то надо прервать выполнение алгоритма и сообщить об ошибке.

Ошибка в процессе выполнения алгоритма возникает, когда невозможно выполнить очередной шаг — например, если не установлено отношение предшествования между двумя сравниваемыми символами (на шагах 2 и 4) или если не удается найти нужное правило в грамматике (на шаге 4). Тогда выполнение алгоритма немедленно прерывается.

Пример 1.

https://studfiles.net/preview/4599549/page:23/https://life-prog.ru/view_psp.php?id=13

Дана грамматика $G(\{a, (,)\}, \{S, R\}, P, S)$, с правилами P:

1) $S \to (R|a$; 2)$R \to Sa)$. Построить распознаватель для строки $(((aa)a)a)\perp_к$.

Шаг 1. Построим множества крайних левых и крайних правых символов $L(A)$ и $R(A)$ относительно всех нетерминальных символов грамматики (*таблица 1.1*).

Шаг	$L_i(A)$	$R_i(A)$
0	$L_0(S)=\{(, a\}$ $L_0(R)=\{S\}$	$R_0(S)=\{R, a\}$ $R_0(R)=\{\}\}$
1	$L_1(S)=\{(, a\}$ $L_1(R)=\{S, (, a\}$	$R_1(S)=\{R, a,)\}$ $R_1(R)=\{\}\}$
2	$L_2(S)=\{(, a\}$ $L_2(R)=\{S, (, a\}$	$R_2(S)=\{R, a,)\}$ $R_2(R)=\{\}\}$
Результат	$L(S)=\{(, a\}$ $L(R)=\{S, (, a\}$	$R(S)=\{R, a,)\}$ $R(R)=\{\}\}$

Таблица 1.1 – Построение множеств L(A) и R(A) для грамматики G

Шаг 2. На основе построенных множеств и правил вывода грамматики составим матрицу предшествования символов (таблица 1.2).

Символы	*S*	*R*	*a*	(	)	$\perp_к$
S			=•			
R			•>			•>
a			•>		=•	•>
(	<•	=•	<•	<•		
)			•>			•>
$\perp_н$			<•	<•		

Таблица 1.2 – Матрица предшествования символов грамматики

В правиле грамматики $S \rightarrow (R$ символ (стоит слева от нетерминального символа *R*. Во множестве *L*(*R*) входят символы *S*, (,*a*. Ставим знак<• в клетках матрицы, соответствующих этим символам, в строке для символа (.

В правиле грамматики $R \rightarrow Sa)$ символ *a* стоит справа от нетерминального символа *S*. Во множество *R*(*S*) входят символы *R*,*a*,). Ставим знак •> в клетках матрицы, соответствующих этим символам, в столбце для символа *a*.

В строке символа $\perp_н$ ставим знак<• в клетках символов, входящих во множество *L*(*S*), т.е. символов (,*a*. В столбце символа $\perp_к$ ставим знак•> в клетках, входящих во множество *R*(*S*), т.е. символов *R*,*a*,).

В клетках, соответствующих строке символа *S* и столбцу символа *a*, ставим знак=·, т.к. существует правило $R \rightarrow Sa)$, в котором эти символы стоят рядом. По тем же соображениям ставим знак=·в клетках строки *a* и столбца), а также строки (и столбца *R*.

Шаг 3. Функционирование распознавателя для цепочки (((aa)a)a) показано в таблице 1.3.

Шаг	Стек	Входной буфер	Действие
1	$\perp_н$	$(((aa)a)a)\perp_к$	сдвиг
2	$\perp_н($	$((aa)a)a)\perp_к$	сдвиг
3	$\perp_н(($	$(aa)a)a)\perp_к$	сдвиг
4	$\perp_н(((a$	$aa)a)a)\perp_к$	сдвиг
5	$\perp_н(((a$	$a)a)a)\perp_к$	свертка $S\rightarrow a$
6	$\perp_н(((S$	$a)a)a)\perp_к$	сдвиг
7	$\perp_н(((Sa$	$)a)a)\perp_к$	сдвиг
8	$\perp_н(((Sa)$	$a)a)\perp_к$	свертка $R\rightarrow Sa)$
9	$\perp_н(((R$	$a)a)\perp_к$	свертка $S\rightarrow(R$
10	$\perp_н((S$	$a)a)\perp_к$	сдвиг
11	$\perp_н((Sa$	$)a)\perp_к$	сдвиг

12	$\perp_н((Sa)$	$a)\perp_к$	свертка $R\rightarrow Sa)$
13	$\perp_н((R$	$a)\perp_к$	свертка $S\rightarrow(R$
14	$\perp_н(S$	$a)\perp_к$	сдвиг
15	$\perp_н(Sa$	$)\perp_к$	сдвиг
16	$\perp_н(Sa)$	$\perp_к$	свертка $R\rightarrow Sa)$
17	$\perp_н(R$	$\perp_к$	свертка $S\rightarrow(R$
18	$\perp_н S$	$\perp_к$	строка принята

Таблица 1.3. – Алгоритм работы распознавателя цепочки $(((aa)a)a)$

Шаг 4. Получили следующую цепочку вывода:

$S\Rightarrow(R\Rightarrow(Sa)\Rightarrow((Ra)\Rightarrow((Sa)a)\Rightarrow(((Ra)a)\Rightarrow(((Sa)a)a)\Rightarrow(((aa)a)a)$.

Пример 2.

I -> TR | T
R -> +T | -T | +TR | -TR
T -> a|b

Построим крайние левые и правые:

1. L(I) = { T}
2. L(R) = { +,-}
3. L(T) = { a,b}
4. R(I) = { R,T}
5. R(R)= {T,R}
6. R(T) = {a,b}

Дополняем к T входящие a,b:

7. L(I) = { T,a,b}

8. L(R) = { +,-}
9. L(T) = { a,b}
10. R(I) = { R,T,a,b}
11. R(R) = {T,R, a,b}
12. R(T) = {a,b}

	T	R	+	-	a	b	Ik
T		=•	<•	<•			•>
R							•>
+	=•				<•	<•	
-	=•				<•	<•	
a		•>	•>	•>			•>
b		•>	•>	•>			•>
In	<•				<•	<•	

Процесс распознавания работает как LR анализатор:

a+b-a	In	П
a+b-a	Ina	C
+b-a	InT	П
b-a	InT+	П
-a	InT+b	C
-a	InT+T	П
a	InT+T-	П
Ik	InT+T-a	C
Ik	InT+T-T	C
Ik	InT+TR	C
Ik	InTR	C
Ik	InI	C

Пример 3.

I -> TS;
T -> int | char
S -> *BR | BR | B
R -> , S
B -> i | j

Построим крайние левые и правые:

L(I) = { T, int ,char}
L(T) = {int, char}
L(S) = {*, B, i,j}
L(R) = {,}

L(B) = {i,j}
R(I) = {;}
R(T) = {int, char}
R(S) = {R,B, i, j,S}
R(R) = {S, R, B, i,j}
R(B) = {I,j}
Строим таблицы:

	T	S	;	int	ch ar	*	B	R	,	i	j	Ik
T		=•				<•	<•			<•	<•	
S			=• •>									
;												
Int		•>				•>	•>			•>	•>	
Ch ar		•>				•>	•>			•>	•>	
*							=•			<•	<•	
B			•>					=•	<•			
R			•>									
,												
I			•>					•>	•>			
J			•>					•>	•>			
In	<•			<•	<•							

В таблице имеет два предшествования. Это на пересечении ; и S.
Следовательно правило нужно переписать:
I -> TS;
T -> int | char
S -> ,*BR | ,BR | ,B
B -> i | j
Тогда буква S должна исчезнуть.
R(R) = {R, B, i,j}

Грамматики операторного предшествования

Операторной грамматикой называется КС-грамматика без λ-правил, в которой правые части всех правил не содержат смежных нетерминальных символов. Для операторной грамматики отношения предшествования можно задать на множестве терминальных символов (включая символы $\perp_H$ и $\perp_K$).

Грамматикой операторного предшествования называется операторная КС-грамматика G(**VN,VT,P**,S), **V** = **VT**∪**VN**, для которой выполняются следующие условия:

1. Для каждой упорядоченной пары терминальных символов выполняется не более чем одно из трех отношений предшествования:

- a =• Ь, если и только если существует правило A→xaby ∈ **P** или правило A→xaCby, где a,b∈**VT**, A,C∈**VN**, x,y∈**V***;

- a <• b, если и только если существует правило A→xaCy ∈ **P** и вывод C⇒*bz или вывод C ⇒*Dbz, где a,b∈**VT**, A,C,D∈**VN**, x,y,z∈**V***;
- a •> b, если и только если существует правило A→xCby ∈ **P** и вывод C⇒*za или вывод C ⇒*zaD, где a,b∈**VT**, A,C,D∈**VN**, x,y,z∈**V***.

2. Различные порождающие правила имеют разные правые части, λ-правила отсутствуют.

Отношения предшествования для грамматик операторного предшествования определены таким образом, что для них выполняется еще одна особенность — правила грамматики операторного предшествования не могут содержать двух смежных нетерминальных символов в правой части. То есть в грамматике операторного предшествования G(**VN,VT,P**,S), **V** = **VT**∪**VN** не может быть ни одного правила вида: A→xBCy, где A,B,C∈**VN**, x.y∈**V*** (здесь x и y — это произвольные цепочки символов, могут быть и пустыми).

Для грамматик операторного предшествования также известны следующие свойства:

- всякая грамматика операторного предшествования задает детерминированный КС-язык (но не всякая грамматика операторного предшествования при этом является однозначной!);
- легко проверить, является или нет произвольная КС-грамматика грамматикой операторного предшествования (точно так же, как и для простого предшествования).

Как и для многих других классов грамматик, для грамматик операторного предшествования не существует алгоритма, который бы мог преобразовать произвольную КС-грамматику в грамматику операторного предшествования (или доказать, что преобразование невозможно).

Принцип работы распознавателя для грамматики операторного предшествования аналогичен грамматике простого предшествования, но отношения предшествования проверяются в процессе разбора только между терминальными символами.

Для грамматики данного вида на основе установленных отношений предшествования также строится матрица предшествования, но она содержит только терминальные символы грамматики.

Для построения этой матрицы удобно ввести множества крайних левых и крайних правых терминальных символов относительно нетерминального символа A - $\mathbf{L}^t(A)$ или $\mathbf{R}^t(A)$:

$\mathbf{L}^t(A)$ = {t | ∃ A⇒*tz или ∃ A⇒*Ctz }, где t∈**VT**, A,C∈**VN**, z∈**V***;

$\mathbf{R}^t(A)$ = {t | ∃ A⇒*zt или ∃ A⇒*ztC }, где t∈**VT**, A,C∈**VN**, z∈**V***.

Тогда определения отношений операторного предшествования будут выглядеть так:

- a =• Ь, если ∃ правило A→xaby ∈ **P** или правило U→xaCby, где a,b∈**VT**, A,C∈**VN**, x,y∈**V***;
- a <• Ь, если ∃ правило A->xaCy ∈ **P** и b∈$\mathbf{L}^t(C)$, где a,b∈**VT**, A,C∈**VN**, x,y∈**V***;

- a •> b, если ∃ правило A->xCby ∈ **P** и a∈$\mathbf{R}^t$(C), где a.b∈**VT**, A.C∈**VN**, x,y∈**V***.

В данных определениях цепочки символов x,y,z могут быть и пустыми цепочками.

Для нахождения множеств $\mathbf{L}^t$(A) и $\mathbf{R}^t$(A) предварительно необходимо выполнить построение множеств **L**(A) и **R**(A), как это было рассмотрено ранее. Далее для построения $\mathbf{L}^t$(A) и $\mathbf{R}^t$(A) используется следующий алгоритм:

Шаг 1. ∀ A∈**VN**:

$\mathbf{R}^t_0$(A) = {t | A→ytB или A→yt, t∈**VT**, B∈**VN**, y∈**V***},

$\mathbf{L}^t_0$(A) = {t | A→Bty или A→ty, t∈**VT**, B∈**VN**, y∈**V***}.

Для каждого нетерминального символа A ищем все правила, содержащие A в левой части. Во множество **L**(A) включаем самый левый терминальный символ из правой части правил, игнорируя нетерминальные символы, а во множество **R**(A) — самый крайний правый терминальный символ из правой части правил. Переходим к шагу 2.

Шаг 2. ∀ A∈**VN**:

$\mathbf{R}^t_i$(A) = $\mathbf{R}^t_{i-1}$(A) ∪ $\mathbf{R}^t_{i-1}$(B), ∀ B ∈ (**R**(A) ∩ **VN**),

$\mathbf{L}^t_i$(A) = $\mathbf{L}^t_{i-1}$(A) ∪ $\mathbf{L}^t_{i-1}$(B), ∀ B ∈ (**L**(A) ∩ **VN**).

Для каждого нетерминального символа A: если множество **L**(A) содержит нетерминальные символы грамматики A', A", …, то его надо дополнить символами, входящими в соответствующие множества $\mathbf{L}^t$(A'), $\mathbf{L}^t$(A''), ... и не входящими в $\mathbf{L}^t$(A). Ту же операцию надо выполнить для множеств **R**(A) и $\mathbf{R}^t$(A).

Шаг 3. Если ∃ A∈**VN**: $\mathbf{R}^t_i$(A) ≠ $\mathbf{R}^t_{i-1}$(A) или $\mathbf{L}^t_i$(A) ≠ $\mathbf{L}^t_{i-1}$(A), то i:=i+1 и вернуться к шагу 2, иначе построение закончено: **R**(A) = $\mathbf{R}^t_i$(A) и **L**(A) = $\mathbf{L}^t_i$(A).

Если на предыдущем шаге хотя бы одно множество $\mathbf{L}^t$(A) или $\mathbf{R}^t$(A) для некоторого символа грамматики изменилось, то надо вернуться к шагу 2, иначе построение закончено.

Для практического использования матрицу предшествования дополняют символами $\perp_H$ и $\perp_K$ (начало и конец цепочки). Для них определены следующие отношения предшествования:

$\perp_H$ <• a, ∀ a∈**VT**, если ∃ S⇒*ax или ∃ S⇒*Cax, где S,C∈**VN**, x∈**V*** или если a∈$\mathbf{L}^t$(S);

$\perp_K$ •> a, ∀ a∈**VT**, если ∃ S⇒*xa или ∃ S⇒*xaC, где S,C∈**VN**, x∈**V*** или если a∈$\mathbf{R}^t$(S).

Здесь S — целевой символ грамматики.

Матрица предшествования служит основой для работы распознавателя языка, заданного грамматикой операторного предшествования. Поскольку она содержит только терминальные символы, то, следовательно, будет иметь меньший размер, чем аналогичная матрица для грамматики простого предшествования. Следует отметить, что напрямую сравнивать матрицы двух грамматик нельзя — не всякая грамматика простого предшествования является грамматикой операторного предшествования, и наоборот. Например, рассмотренная далее в примере грамматика операторного предшествования не

является грамматикой простого предшествования (читатели могут это проверить самостоятельно). Однако если две грамматики эквивалентны и задают один и тот же язык, то их множества терминальных символов должны совпадать. Тогда можно утверждать, что размер матрицы для грамматики операторного предшествования всегда будет меньше, чем размер матрицы эквивалентной ей грамматики простого предшествования.

Все, что было сказано выше о способах хранения матриц для грамматик простого предшествования, в равной степени относится также и к грамматикам операторного предшествования, с той только разницей, что объем хранимой матрицы будет меньше.

Алгоритм «сдвиг-свертка» для грамматики операторного предшествования

Алгоритм выполняется расширенным МП-автоматом и имеет те же условия завершения и обнаружения ошибок. Основное отличие состоит в том, что при определении отношения предшествования этот алгоритм не принимает во внимание находящиеся в стеке нетерминальные символы и при сравнении ищет ближайший к верхушке стека терминальный символ. Однако после выполнения сравнения и определения границ основы при поиске правила в грамматике нетерминальные символы следует, безусловно, принимать во внимание.

Алгоритм состоит из следующих шагов.

Шаг 1. Поместить в верхушку стека символ $\perp_{\text{Н}}$, считывающую головку — в начало входной цепочки символов.

Шаг 2. Сравнить с помощью отношения предшествования терминальный символ, ближайший к вершине стека (левый символ отношения), с текущим символом входной цепочки, обозреваемым считывающей головкой (правый символ отношения). При этом из стека надо выбрать самый верхний терминальный символ, игнорируя все возможные нетерминальные символы.

Шаг 3. Если имеет место отношение **<•** или **=•**, то произвести сдвиг (перенос текущего символа из входной цепочки в стек и сдвиг считывающей головки на один шаг вправо) и вернуться к шагу 2. Иначе перейти к шагу 4.

Шаг 4. Если имеет место отношение **•>**, то произвести свертку. Для этого надо найти на вершине стека все терминальные символы, связанные отношением **=•** («основу»), а также все соседствующие с ними нетерминальные символы (при определении отношения нетерминальные символы игнорируются). Если терминальных символов, связанных отношением **=•**, на верхушке стека нет, то в качестве основы используется один, самый верхний в стеке терминальный символ стека. Все (и терминальные, и нетерминальные) символы, составляющие основу, надо удалить из стека, а затем выбрать из грамматики правило, имеющее правую часть, совпадающую с основой, и поместить в стек левую часть выбранного правила. Если правило, совпадающее с основой, найти не удалось, то необходимо прервать выполнение алгоритма и сообщить об ошибке, иначе, если разбор не закончен, то вернуться к шагу 2.

Шаг 5. Если не установлено ни одно отношение предшествования между текущим символом входной цепочки и самым верхним терминальным

символом в стеке, то надо прервать выполнение алгоритма и сообщить об ошибке.

Конечная конфигурация данного МП-автомата совпадает с конфигурацией при распознавании цепочек грамматик простого предшествования.

Пример

Дано: $G(\{(,), \wedge, \&, \sim, a\}, \{S, T, E, F\}, P, S)$, где

P: $S \to S\wedge T \mid T$

$T \to T\&E \mid E$

$E \to \sim E \mid F$

$F \to (E) \mid a$

Построить: распознаватель для G.

Построение множеств $L(A)$ и $R(A)$:

i	$L_i(A)$	$R_i(A)$
0	$L_0(S)=\{S, T\}$ $L_0(T)=\{T, E\}$ $L_0(E)=\{\sim, F\}$ $L_0(F)=\{(, a\}$	$R_0(S)=\{T\}$ $R_0(T)=\{E\}$ $R_0(E)=\{E, F\}$ $R_0(F)=\{), a\}$
1	$L_1(S)=\{S, T, E\}$ $L_1(T)=\{T, E, \sim, F\}$ $L_1(E)=\{\sim, F, (, a\}$ $L_1(F)=\{(, a\}$	$R_1(S)=\{T, E\}$ $R_1(T)=\{E, F\}$ $R_1(E)=\{E, F,), a\}$ $R_1(F)=\{), a\}$
2	$L_2(S)=\{S, T, E, \sim, F, (, a\}$ $L_2(T)=\{T, E, \sim, F, (, a\}$ $L_2(E)=\{\sim, F, (, a\}$ $L_2(F)=\{(, a\}$	$R_2(S)=\{T, E, F,), a\}$ $R_2(T)=\{E, F,) a\}$ $R_2(E)=\{E, F,), a\}$ $R_2(F)=\{), a\}$
3	$L_3(S)=\{S, T, E, \sim, F, (, a\}$ $L_3(T)=\{T, E, \sim, F, (, a\}$ $L_3(E)=\{\sim, F, (, a\}$ $L_3(F)=\{(, a\}$	$R_3(S)=\{T, E, F,), a\}$ $R_3(T)=\{E, F,) a\}$ $R_3(E)=\{E, F,), a\}$ $R_3(F)=\{), a\}$

Построение множеств $L^t(A)$ и $R^t(A)$:

i	$L^t(A)$	$R^t(A)$
0	$L^t_0(S)=\{\wedge\}$ $L^t_0(T)=\{\&\}$ $L^t_0(E)=\{\sim\}$ $L^t_0(F)=\{(, a\}$	$R^t_0(S)=\{\wedge\}$ $R^t_0(T)=\{\&\}$ $R^t_0(E)=\{\sim\}$ $R^t_0(F)=\{), a\}$
1	$L^t_1(S)=\{\wedge, \&, \sim, (, a\}$ $L^t_1(T)=\{\&, \sim, (, a\}$ $L^t_1(E)=\{\sim, (, a\}$ $L^t_1(F)=\{(, a\}$	$R^t_1(S)=\{\wedge, \&, \sim,), a\}$ $R^t_1(T)=\{\&, \sim,), a\}$ $R^t_1(E)=\{\sim,), a\}$ $R^t_1(F)=\{), a\}$

2	$L^t_2(S)$={^, &, ~, (, *a* } $L^t_2(T)$={&, ~, (, *a*} $L^t_2(E)$={~, (, *a*} $L^t_2(F)$={(, *a*}	$R^t_2(S)$={^, &, ~,), *a*} $R^t_2(T)$={&, ~,), *a*} $R^t_2(E)$={~,), *a*} $R^t_2(F)$={), *a*}

Матрица операторного предшествования символов грамматики :

Символы	^	&	~	*(*	*)*	*a*	$\perp_{\hat{e}}$
^	•>	<•	<•	<•	•>	<•	•>
&	•>	•>	<•	<•	•>	<•	•>
~	•>	•>	<•	<•	•>	<•	•>
(	<•	<•	<•	<•	=•	<•	
)	•>	•>			•>		•>
a	•>	•>			•>		•>
$\perp_{\acute{i}}$	<•	<•	<•	<•		<•	

Для ^ находящейся в правиле вывода слева от нетерминала *T*, во множество $L^t(T)$ входят символы &, ~, (, *a* , значит в строке матрицы для ^ ставим знак меньшего предшествования в позициях этих символов. С другой стороны этот символ ^ находится справа от *S*. Во множество $R^t(S)$ входят символы ^, &, ~,), *a,* значит знак большего предшествования ставится в столбце для ^ в позициях этих символов. Символы (и) в правиле вывода находятся радом, поэтому в позиции этих символов ставится знак равного предшествования (игнорируя нетерминал *E*).

++++++++++++++++ Было

Алгоритм 1.4.13. Пусть G = (T, V, P, S) - КС-грамматика, правила которой пронумерованы числами от 1 до p.

Алгоритм типа “перенос-свертка” для грамматики G назовем пару функций λ = (f, g), где f – называется функцией переноса, а g – функцией свертки. Функции f и g определяются следующим образом:

1. f отображает $(V \cup T \cup \{\perp\})^* \times (T \cup \{\perp\})^*$ в множество {ПЕРЕНОС, СВЕРТКА, ОШИБКА, ДОПУСК}.

2. g отображает $(V \cup T \cup \{\perp\})^* \times (T \cup \{\perp\})^*$ в множество {1,…, ОШИБКА}. Если g(α,w) = i, то правая часть i-го правила является суффиксом цепочки α.

Пусть имеется сентенциальная форма $\alpha X_1 X_2 \beta$, где X_1, $X_2 \in V \cup T$. На некотором этапе разбора либо символ X_1, либо символ X_2, либо они вместе должны войти в основу.

Отношения (•>), (•=) и (<•) называются отношениями предшествования.

Рассмотрим три следующих случая:

X_1 - часть основы, символ X_2 не входит в основу. Символ X_1 будет свернут раньше X_2. В этом случае говорят, что символ X_1 предшествует X_2, и записывают как это как X_1 •> X_2. X_1 – последний символ основы (хвост основы),

т.е. последний символ правой части некоторого правила, а символ X_2 – терминальный символ.

X_1 и X_2 входят в основу, т.е. сворачиваются одновременно, записывают как $X_1 \bullet= X_2$. Символы X_1 и X_2 входят в правую часть некоторого правила грамматики.

X_2 - часть основы, а символ X_1 не входит в основу. Символ X_2 – должен быть свернут раньше X_1, записывают как $X_1 <\bullet X_2$. X_1 – это первый символ основы (головы основы), т.е. первый символ правой части некоторого правила.

Работу алгоритма типа "перенос-свертка" удобно описывать в терминах конфигураций вида ($\perp X_1 \ldots X_m, a_1 \ldots a_n, p_1 \ldots p_r$), где :

$\perp X_1 \ldots X_m$ – содержимое магазина, X_m – верхний символ магазина и $X_i \in V \cup T$, символ ($\perp$) для магазина;

$a_1 \ldots a_n$ – оставшаяся непрочитанная часть входной цепочки, a_1 – текущий входной символ;

$p_1 \ldots p_r$ – разбор, полученный к данному моменту времени.

Один шаг алгоритма λ можно описать с помощью двух отношений: ($\vdash^S$)(перенос) и ($\vdash^r$) (свертка), определенных на конфигурациях следующим образом:

1. Если f(α,aw) = ПЕРЕНОС (П), то входной символ переносится в верхушку магазина и читающая головка сдвигается на один символ вправо. В терминах конфигураций этот процесс описывается так:

$(\alpha, aw, \pi) \vdash^S (\alpha a, w, \pi)$ для $\alpha \in (V \cup T \cup \{\perp\})^*$, $w \in (T \cup \{\varepsilon\})^*$ и $\pi \in \{1,\ldots,p\}^*$.

2. Если f(αβ, w) = СВЕРТКА (С), g(αβ, w) = i и $A \to \beta$ – правило грамматики с номером i, то цепочка β заменяется правой частью правила с номером i, а его номер помещается на выходную ленту, т.е. $(\alpha, aw, \pi) \vdash^r (\alpha A, w, \pi i)$.

3. Если f(α, w) = ДОПУСК, то $(\alpha, w, \pi) \vdash^S$ ДОПУСК (Д).

В остальных случаях $(\alpha, w, \pi) \vdash^S$ ОШИБКА (пустое значение в таблице).

Отношение ($\vdash$) определяется как объединение отношений ($\vdash^S$) и ($\vdash^r$), а транзитивное замыкание отношений ($\vdash^+$) и ($\vdash^*$) определяется как обычно.

Для $w \in T^*$ будем записывать $\lambda(w) = \pi$, если $(\perp, w, \varepsilon) \vdash^* (\perp S, \varepsilon, \pi) \vdash^S$ ДОПУСК, и $\lambda(w)$ = ОШИБКА, в противном случае.

Для практического прменения алгоритм мало пригоден, так как для определения функции переноса он требует анализа всей цепочки в магазине и всей необработанной части входной цепочки. Аналогичный недостаток имеет алгоритм и при определении функции свертки. В литературе описано несколько вариантов грамматик предшествования.

Для всех классов грамматик предшествования общим является способ поиска правого конца основы (*хвоста* основы). При разборе алгоритмом "перенос-свертка", всякий раз, когда между верхним символом магазина и первым из необработанных входных символов выполняется отношение $\bullet>$, принимается решение о свертке, в противном случае делается перенос.

Таким образом, с помощью отношения $\bullet>$ локализуется правый конец основы правовыводимой цепочки. Определение левого конца основы (*головы* основы) и нужной свертки осуществляется в зависимости от используемого типа отношения предшествования.

3.9. Пример алгоритм “перенос-свертка”

1. Применить алгоритм типа “перенос-свертка” для заданной грамматики G= (T, V, P, S), где P = {p_1,p_2,p_3,p_4}, p_1) S→ bAb, p_2) A→ cB, p_3) A→ a, p_4) B→ Aad

2. Описать работу алгоритма.

Функция переноса

f	ax	Bx	cx	dx	ε
αA	П	П			
αB	С	С			
Aa	С	С		П	
Ab	П		П		С
Ac	П		П		
Ad	С	С			
⊥		П			
^S					Д

Функция свертки

g	ax	bx	x	ε
⊥b Ab				1
αA ad		4		
αc B	2	2		
Aa			3	

2. Разберем с помощью построенного алгоритма λ входную цепочку bcaadb.

Начальная конфигурация алгоритма (⊥, bcaadb, ε).

Шаг 1. Выполнение алгоритма определяется значением f(⊥, bcaadb), которое, как видно из определения функции f, имеет значение ПЕРЕНОС. Алгоритм переходит в конфигурацию (⊥b,caadb, ε), выполняя шаг:

(⊥, bcaadb, ε) $\vdash^r$ (⊥b,caadb, ε) и следующие шаги:

(⊥b,caadb, ε) $\vdash^r$ (⊥bc, aadb, ε) $\vdash^r$ (⊥bca, adb, ε).

Шаг 2. Очередной шаг выполнения алгоритма определяется значением f(⊥bca, adb, ε), котрое имеет значение СВЕРТКА. Правило, используемое при свертке, определяется значением функции g(⊥bca, adb) = 3 и переходит в следующую конфигурацию: (⊥bca, adb, ε) $\vdash^s$ (⊥bcA, adb, 3), иследующие шаги: (⊥bcA, adb, 3) $\vdash^r$ (⊥bcAa, db, 3) $\vdash^r$ (⊥bcAad, b, 3) $\vdash^s$ (⊥bcB, b, 34)

$\vdash^s$ (⊥bA, b, 342) $\vdash^r$ (⊥bAb, ε, 342) $\vdash^r$ (⊥S, ε, 3421) $\vdash^S$ ДОПУСК.

Таким образом, λ(bcaadb) = 3421, правый вывод цепочки bcaadb должен быть равен 1243. Для проверки построим цепочку по заданному правому выводу:

S ⇒ (p_1) bAb ⇒ (p_2) bcBb ⇒ (p_4) bcAdb ⇒ (p_3) bcaadb

++++++++++++++++++++++++++++++

Глава 4. Грамматики общего вида и машины Тьюринга

4.1. Грамматики общего вида и машина Тьюринга

Определение 15. Грамматика G = (T, V, P, S) называется *грамматикой типа 0*, если на правила вывода не накладывается никаких ограничений (кроме тех, которые указаны в определении грамматики).

Грамматики типа 0 - это самые общие грамматики.

Языки высокого уровня не являются рекурсивными. Проблема выявления принадлежности выражения w языку, порожденному грамматикой типа 0, где $w \in T^*$, в общем случае неразрешима.

Определение 16. Машина Тьюринга (служит для проверки разрешимости алгоритма), в иерархии Хомского занимает самый верхний уровень:

$M=(Q, \Sigma, R, L, B, \delta, q_0,)$

Q - множество состояний

Σ - входной алфавит

R, L - правые и левые символы, не входящие в $Q \cup \Sigma$, B - пробел ($B \in \Sigma$)

δ - множество правил **переписывания**

q_0 - исходное состояние

$\delta = q_i a_i \rightarrow q_k a_l$, a_i - входные символы $q_i a_i \rightarrow q_k$ { **B**,L,R},

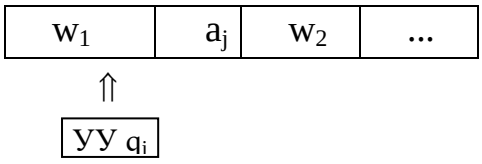
$\delta = Q \times \Sigma$ в $Q \times (\Sigma \cup \{\mathbf{B}, L, R\})$ $q_i a_j \rightarrow q_k \{\mathbf{B}, L, R\}$

Машина Тьюринга - детерминированный преобразователь. Конфигурация машины - множество элементов следующего вида:

$\Sigma^* \times Q \times \Sigma^+$, то есть последовательности $w_1 q_1 a_j w_2$, где w_1 и $w_2 \in \Sigma^*$, $a_j \in \Sigma$, $q_i \in Q$.

Правила из δ задают последовательные переходы от одной конфигурации (до тех пор, пока не будет достигнута заключительная конфигурация).

Конфигурация $w_1 q_i a_j w_2$ называется заключительной, если к ней неприменимо ни одно из правил δ (т.е. правил с заголовком $q_1 a_j$).



Работа машины сводится к следующему, управляющее устройство (УУ):

1. Читает содержимое ячейки, обозреваемой головкой в текущий момент времени.
2. Пишет в обозреваемую ячейку соответствующий символ из входного словаря.
3. Перемещается в соседнюю ячейку, находящуюся слева или справа от обозреваемой ячейки.

Находясь в q_i читает символ a_i и переходит в состояние q_k и либо печатает в этой ячейке символ a_i, либо перемещает головку влево или вправо.

Если к каждой цепочке применить правило, то получится новая конфигурация. Воспринимаемый машиной Тьюринга язык $L=\{w \in \Sigma^* | \ q_0$ и $w \Rightarrow^* C_f$, где C_f - заключительная конфигурация.$\}$

Если w- цепочка языка, то машина за конечный период времени находит заключительную конфигурацию C_f. Если же $w \notin L$, то машина не останавливается, что приводит к неразрешимости проблемы распознавания принадлежности данного выражения языку.

4.2. Контекстно-зависимые грамматики и ленточные автоматы

Определение 17. Грамматика G = (T, V, P, S) называется *неукорачивающей грамматикой*, если каждое правило из P имеет вид $\alpha \rightarrow \beta$, где $\alpha \subset (T \cup V)^+$, $\beta \subset (T \cup V)^+$ и $|\alpha| \leq |\beta|$.

Определение 18. Грамматика G = (T, V, P, S) называется *контекстно-зависимой (КЗ),* если каждое правило из P имеет вид $\alpha \rightarrow \beta$, где $\alpha = \xi_1 A \xi_2$; $\beta = \xi_1 \gamma \xi_2$; $A \subset V$; $\gamma \subset (T \cup V)^+$; $\xi_1, \xi_2 \subset (T \cup V)^*$.

Грамматику типа 1 можно определить как неукорачивающую либо как контекстно-зависимую.

Выбор определения не влияет на множество языков, порождаемых грамматиками этого класса, поскольку доказано, что множество языков, порождаемых неукорачивающими грамматиками, совпадает с множеством языков, порождаемых КЗ-грамматиками.

5. Соотношения между классами грамматик и языками

Отношения между классами КС-грамматик

В общем случае можно выделить правоанализируемые и левоанализируемые КС-грамматики. Об этих двух принципиально разных классах грамматик уже говорилось выше: первые предполагают построение левостороннего (восходящего) распознавателя, вторые — правостороннего (нисходящего). Это вовсе не значит, что для КС-языка, заданного, например, некоторой левоанализируемой грамматикой, невозможно построить расширенный МП-автомат, который порождает правосторонний вывод. Указанное разделение грамматик относится только к построению на их основе детерминированных МП-автоматов и детерминированных расширенных МП-автоматов. Только эти типы автоматов представляют интерес при создании компиляторов и анализе входных цепочек языков программирования. Недетерминированные автоматы, порождающие как левосторонние, так и правосторонние выводы, можно построить в любом случае для языка заданного любой КС-грамматикой, но для создания компилятора такие автоматы интереса не представляют (см. раздел «Распознаватели КС-языков. Автоматы с магазинной памятью»).

На рис. 12.10 изображена условная схема, дающая представление о соотношении классов левоанализируемых и правоанализируемых КС-грамматик [5, 6, т. 2,42, 65].

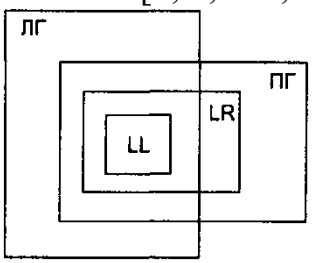


Рис. 12.10. Соотношение классов левоанализируемых и правоанализируемых КС-грамматик

Интересно, что классы левоанализируемых и правоанализируемых грамматик являются несопоставимыми. То есть существуют левоанализируемые КС-грамматики, на основе которых нельзя построить детерминированный расширенный МП-автомат, порождающий правосторонний вывод; и наоборот — существуют правоанализируемые КС-грамматики, не допускающие построение МП-автомата, порождающего левосторонний вывод. Конечно, существуют грамматики, подпадающие под оба класса и допускающие построение детерминированных автоматов как с правосторонним, так и с левосторонним выводом.

Следует помнить также, что все упомянутые классы КС-грамматик — это счетные, но бесконечные множества. Нельзя построить и рассмотреть все возможные левоанализируемые грамматики или даже все возможные LL(1)-грамматики. Сопоставление классов КС-грамматик производится исключительно на основе анализа структуры их правил. Только на основании такого рода анализа произвольная КС-грамматика может быть отнесена в тот или иной класс (или несколько классов).

Все это тем более интересно, если вспомнить, что рассмотренный в данном курсе класс левоанализируемых LL-грамматик является собственным подмножеством класса LR-грамматик: любая LL-грамматика является LR-грамматикой, но не наоборот - существуют LR-грамматики, которые не являются LL-грамматиками. Этот факт также нашел свое отражение в схеме на рис. 12.10. Значит, любая LL-грамматика является правоанализируемой, но существуют также и другие левоанализируемые грамматики, не попадающие в класс правоанализируемых грамматик.

Для LL(k)-грамматик, составляющих класс LL-грамматик, интересна еще одна особенность: доказано, что всегда существует язык, который может быть задан LL(k)-грамматикой для некоторого $k > 0$, но не может быть задан LL(k-1)-грамматикой. Таким образом, все LL(k)-грамматики для всех k представляют определенный интерес (другое дело, что распознаватели для них при больших значениях k будут слишком сложны). Интересно, что проблема эквивалентности для двух LL(k)-грамматик разрешима.

С другой стороны, для LR(k)-грамматик, составляющих класс LR-грамматик, доказано, что любой язык, заданный LR(k)-грамматикой с $k > 1$, может быть задан LR(1)-грамматикой. То есть LR(k)-грамматики с $k > 1$ интереса не представляют. Однако доказательство существования LR(1)-грамматики вовсе не означает, что такая грамматика всегда может быть построена (проблема преобразования КС-грамматик неразрешима).

На рис. 12.11 условно показана связь между некоторыми классами КС-грамматик, упомянутых в данном пособии. Из этой схемы видно, например, что любая LL-грамматика является LR-грамматикой, но не всякая LL-грамматика является LR(1)-грамматикой.

Рис. 12.11. Схема взаимосвязи некоторых классов КС-грамматик

Если вспомнить, что любой детерминированный КС-язык может быть задан, например, LR(1)-грамматикой, но в то же время, классы левоанализируемых и правоанализируемых грамматик несопоставимы, то напрашивается вывод: один и тот же детерминированный КС-язык может быть задан двумя или более несопоставимыми между собой грамматиками. Таким образом, можно вернуться к мысли о том, что проблема преобразования КС-грамматик неразрешима (на самом деле, конечно, наоборот: из неразрешимости проблемы преобразования КС-грамматик следует возможность задать один и тот же КС-язык двумя несопоставимыми грамматиками). Это, наверное, самый интересный вывод, который можно сделать из сопоставления разных классов КС-грамматик.

Отношения между классами КС-языков

КС-язык называется языком некоторого класса КС-языков, если он может быть задан КС-грамматикой из данного класса КС-грамматик. Например, класс LL-языков составляют все языки, которые могут быть заданы с помощью LL-грамматик.

Соотношение классов КС-языков представляет определенный интерес, оно не совпадает с соотношением классов КС-грамматик. Это связано с многократно уже упоминавшейся проблемой преобразования грамматик. Например, выше уже говорилось о том, что любой LL-язык является и LR(1)-языком — то есть язык, заданный LL-грамматикой, может быть задан также и LR(1)-грамматикой. Однако не всякая LL-грамматика является при этом LR(1)-грамматикой и не всегда можно найти способ, как построить LR(1)-грамматику, задающую тот же самый язык, что и исходная LL-грамматика.

На рис. 12.12 приведено соотношение между некоторыми известными классами КС-языков [6, т. 2, 42, 47].

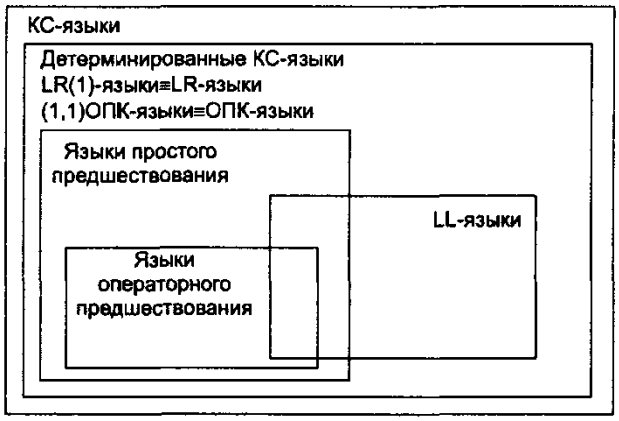


Рис. 12.12. Соотношение между различными классами КС-языков

Следует обратить внимание прежде всего на то, что интересующий разработчиков компиляторов в первую очередь класс детерминированных КС-языков полностью совпадает с классом LR-языков и, более того, совпадает с классом LR(1)-языков. То есть доказано, что для любого детерминированного КС-языка существует задающая его LR(1)-грамматика. Этот факт уже упоминался выше. Проблема состоит в том, что не всегда возможно найти такую грамматику и нет формализованного алгоритма, как ее построить в общем случае.

Также уже упоминалось, что LL-языки являются собственным подмножеством LR-языков: всякий LL-язык является одновременно LR-языком, но существуют LR-языки, которые не являются LL-языками. Поэтому LL-языки образуют более узкий класс, чем LR-языки.

Языки простого предшествования, в свою очередь, также являются собственным подмножеством LR-языков, а языки операторного предшествования — собственным подмножеством языков простого предшествования. Интересно, что языки операторного предшествования представляют собой более узкий класс, чем языки простого предшествования.

В то же время языки простого предшествования и LL-языки несопоставимы между собой: существуют языки простого предшествования, которые не являются LL-языками, и в то же время существуют LL-языки, которые не являются языками простого предшествования. Однако существуют языки, которые одновременно являются и языками простого предшествования, и LL-языками. Аналогичное замечание относится также к соотношению между собой языков операторного предшествования и LL-языков.

Можно еще отметить, что язык арифметических выражений над символами a и Ь, заданный грамматикой G({+.-,/,*,a,b},{S,T,E},**P**,S), **P** = {S→S+T|S-T|T, T→T*E|T/E|E, E→(S)|a|b), который многократно использовался в примерах в данном учебном пособии, подпадает под все указанные выше классы языков. Из приведенных ранее примеров можно заключить, что этот язык является и LL-языком, и языком операторного предшествования, а следовательно, и языком простого предшествования и, конечно, LR(1)-языком. В то же время этот язык по мере изложения материала пособия описывался различными грамматиками, не все из которых могут быть отнесены в указанные классы. Более того, он может быть задан с помощью грамматики, которая не являлась даже однозначной.

Таким образом, соотношение классов КС-языков не совпадает с соотношением задающих их классов КС-грамматик. Это связано с неразрешимостью проблем преобразования и эквивалентности грамматик, которые не имеют строго формализованного решения.

Соотношения между типами грамматик:

1) любая регулярная грамматика является КС-грамматикой;

2) любая регулярная грамматика является укорачивающей КС-грамматикой (УКС). Отметим, что УКС-грамматика, содержащая правила вида $A \to \varepsilon$, не является КЗ-грамматикой и не является неукорачивающей грамматикой;

3) любая (приведенная) КС-грамматика является КЗ-грамматикой;

4) любая (приведенная) КС-грамматика является неукорачивающей грамматикой;

5) любая КЗ-грамматика является грамматикой типа 0.

6) любая неукорачивающая грамматика является грамматикой типа 0.

Определение 19. Язык L(G) является *языком типа k*, если его можно описать грамматикой типа k.

Соотношения между типами языков:

1) каждый регулярный язык является КС-языком, но существуют КС-языки, которые не являются регулярными (например, $L = \{a^n b^n \mid n>0\}$);

2) каждый КС-язык является КЗ-языком, но существуют КЗ-языки, которые не являются КС-языками (например, $L = \{a^n b^n c^n \mid n>0\}$);

3) каждый КЗ-язык является языком типа 0. УКС-язык, содержащий пустую цепочку, не является КЗ-языком. Если язык задан грамматикой типа k, то это не значит, что не существует грамматики типа k' (k'>k), описывающей тот же язык. Поэтому, когда говорят о языке типа k, обычно имеют в виду максимально возможный номер k.

Например, КЗ-грамматика $G_1 = (\{0,1\}, \{A,S\}, P_1, S)$ и КС-грамматика $G_2 = (\{0,1\}, \{S\}, P_2, S)$, где $P_1 = \{S \to 0A1,\ 0A \to 00A1, A \to \varepsilon\}$ и $P_2 = \{S \to 0S1 \mid 01\}$

описывают один и тот же язык $L = L(G_1) = L(G_2) = \{ 0^n 1^n \mid n>0\}$. Язык L называют КС-языком, т.к. существует КС-грамматика, его описывающая. Но он не является регулярным языком, т.к. не существует регулярной грамматики, описывающей этот язык.

Глава 5. Методы описания перевода

Грамматики описывают только синтаксические аспекты формальных языков. При построении трансляторов необходимо каждой входной цепочке (программе на входном языке) поставить в соответствие другую цепочку (программу на выходном языке). Отношение между входными и выходными цепочками называют переводом.

Возникает проблема (аналогичная описании синтаксиса языка) - задания бесконечного перевода конечными средствами.

5.1 Перевод и семантика

Существуют два подхода к описанию перевода:

1. использование системы, аналогичной формальной грамматике или являющейся её расширением, которая порождает перевод (точнее пару цепочек, принадлежавших переводу);
2. применение автомата, распознающего входную цепочку и выдающего на выходе перевод этой цепочки.
3. Естественными требованиями к устройству, выполнявшему трансляцию, являются:
4. простота анализа и синтеза: мы должны, с одной стороны, по описанию перевода достаточно легко установить, какие пары цепочек принадлежат переводу, и, с другой стороны, построение описания перевода не должно быть непосильной задачей для разработчика;
5. возможность автоматического построения транслятора по описанию перевода;
6. небольшой объем и эффективность трансляции;
7. корректность и т. п.

Перейдём к рассмотрению простейших способов описания перевода.

Определение. Пусть $\sum$ — входной алфавит и Δ — выходной алфавит. Переводом с языка $L_1 \subseteq \sum^*$ на язык $L_2 \subseteq \Delta^*$ называется отношение τ из $\sum^*$ в Δ^*, для которого L_1 — область определения, а L_2 — множество значений.

Если (x, y) $\in \tau$, то Цепочка y называется выходом для цепочки x. В общем случае в переводе τ для одной входной цепочки может быть задано более одной выходной цепочки.

Для языков программирования перевод должен быть функцией, т. е. для каждой входной программы должно быть не более одной выходной программы.

Простейшие переводы можно задать при помощи гомоморфизма.

Пример 5.11. Пусть требуется перевести символы, образующие целые числа со знаком, в и названия. В этом случае гомоморфизм h можно определить следующим образом:

$\sum$ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -, +};

h(a) = a, если a $\in \sum$;

Для a $\in \sum$ Гомоморфизм h(a) определяется в соответствии с табл. 5.1.

Таблица 5.1. Определение гомоморфизма h(a)

a	**h(a)**	**a**	**h(a)**	**a**	**h(a)**
1	один	5	пять	9	девять
2	два	6	шесть	0	нуль
3	три	7	семь	+	плюс
4	четыре	8	восемь	-	минус

Используя данное определение, перевод входной цепочки "-15", содержащей запись числа -15, будет иметь вид: "минус один пять".

Гомоморфизм позволяет описывать только простейшие переводы. Даже такие простые переводы, как τ_1 = {(x, y) | x — инфиксное выражение и y — префиксная запись выражения x} и τ_2 = {(x, y) | x — инфиксное выражение и y — постфиксная запись выражения x} нельзя задать при помощи гомоморфизма, нужны более мощные формализмы.

5.2 СУ-схемы

Схема синтаксически управляемого (СУ-схема) перевода представляет собой систему, порождающую пары цепочек, принадлежавших переводу. Неформально схема синтаксически управляемого перевода представляет собой грамматику, в которой каждому правилу приписан элемент перевода. При порождении цепочки каждый раз, когда правило участвует в этом процессе, элемент перевода генерирует часть выходной цепочки, соответствую щей части входной цепочки, порождённой этим правилом.

Рассмотрим схему синтаксически управляемого перевода, описывающего перевод скобочного арифметического выражения, порождаемого грамматикой G0, в соответствующую польскую инверсную запись. Схема перевода представлена в табл. 5.2.

Таблица 11.2. СУ-схема перевода

	Правило грамматики	Элемент перевода
(1)	E ⟶ E + T	E = ET +
(2)	E ⟶ T	E = T
(3)	T ⟶ T * P	T = TP *
(4)	T ⟶ P	T = P
(5)	P ⟶ (E)	P = E
(6)	P ⟶ i	P = i

В этой схеме правилу Грамматики E → E + T соответствует элемент

перевода E = ET+, который интерпретируется следующим образом: "Перевод нетерминала E, стоявшего в левсй части элемента перевода, представляет собой перевод, порождённый символом E, стоявшим в правей части этого элемента, за которым следуют перевод, порождаемый символом T, и символ '+' ".

Используя СУ-схему, приведённую в табл. 5.2, определим перевод цепочки a * (b + c).

Замечание: для указания местоположения символов выходной цепочки, являющихся переводом символов входной цепочки, будем считать символы a, b, c значениями терминального символа i грамматики.

В начале определим левый вывод входной Цепочки:

$E \Rightarrow_{(2)} T \Rightarrow_{(3)} T * P \Rightarrow_{(4)} P * P \Rightarrow_{(6)} a * P \Rightarrow_{(5)} a * (E) \Rightarrow_{(1)} a * (E + T) \Rightarrow_{(2)}$
$a * (T + T) \Rightarrow_{(4)} a * (P + T) \Rightarrow_{(6)} a * (b + T) \Rightarrow_{(4)} a * (b + P) \Rightarrow_{(6)}$
$a * (b + c).$

Полученный вывод 23465124646 определяет последовательность использовав- неё элементов перевода при порождении выходной Цепочки, поэтому последовательность выводимых пар цепочек имеет следующий вид:

$(E, E) \Rightarrow_{(2)} (T, T)$
$\Rightarrow_{(3)} (T * P, TP *)$
$\Rightarrow_{(4)} (P * P, PP *)$
$\Rightarrow_{(6)} (a * P, aP *)$
$\Rightarrow_{(5)} (a * (E), a * E)$
$\Rightarrow_{(1)} (a * (E + T), a * ET +)$
$\Rightarrow_{(2)} (a * (T + T), a * TT +)$
$\Rightarrow_{(4)} (a * (P + T), a * PT +)$
$\Rightarrow_{(6)} (a * (b + T), a * bT +)$
$\Rightarrow_{(4)} (a * (b + P), a * bP +)$
$\Rightarrow_{(6)} (a * (b + c), a * bc +).$

Каждая выходная цепочка при порождении части входной цепочки получается путем замены (подходившего) не терминала выходной цепочки путем замены (подходившего) не терминала выходной Цепочки значением соответствующего ему элемента перевода.

Рассмотренная схема перевода относится к классу схем, называемых схемами синтаксически управляемого перевода.

Определение: схемой синтаксически управляемого перевода называется пятёрка T = (N, ∑, Δ, R, S), где:

N — конечное множество нетерминальных символов;

∑ — конечный входной алфавит;

Δ — конечный выходной алфавит;

R — конечное множество правил вида $A \rightarrow \alpha, \beta$, где $\alpha \in (N \cup \sum)^*$,

$\beta \in (N \cup \Delta)^*$ и вхождения не терминалов в цепочку β образуют перестановку вхождений не терминалов в цепочку α;

S — начальный символ ($S \in N$).

По определению, каждому вхождению определённого не терминала в цепочку α соответствует некоторое вхождение этого не терминала в цепочку β. Если некоторый не терминал входит в цепочку α более одного раза, то может возникнуть неоднозначность. Для исключения этого будем пользоваться верхними целочисленными индексами, например в правиле $A \to B^{(1)}CB^2$, $B^2B^{(1)}C$ первой, второй и третьей позиции цепочки B(1)CB(2) соответствуют вторая, третья и первая позиции цепочки $B^2B^{(1)}C$.

Примечание: выводимая пара цепочек СУ-схемы T определяется рекурсивно следующим образом:

1) (S, S) — выводимая пара, в которой символы S соответствуют друг другу.

2) Если $(\alpha A\beta, \alpha' A\beta')$ — выводимая пара, в которой два выделенных вхождения не терминала A соответствуют друг другу, и $A \to \alpha, \beta$ — правило из R, то $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ — выводимая пара.

Вхождения не терминалов в цепочки γ и γ' соответствуют Друг Другу точно так же, как они соответствовали в правиле СУ-схемы. Вхождения не терминалов в цепочки α и β соответствуют вхождениям не терминалов в цепочки α' и β' в новой выводимой паре точно так же, как они соответствовали в старой выводимой паре. При необходимости это соответствие будет указываться верхними индексами.

Если между парами цепочек $(\alpha A\beta, \alpha' A\beta')$ и $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ установлена описанная ранее связь, то будем писать $(\alpha A\beta, \alpha'\gamma'\beta') \Rightarrow_T (\alpha\gamma\beta, \alpha'\gamma'\beta')$.

Транзитивное замыкание, рефлексивно-транзитивное замыкание и k-Ю степень отношения $\Rightarrow_T$ будем обозначать как $\Rightarrow^+_T$, $\Rightarrow^*_T$, и $\Rightarrow^k_T$, соответственно. В очевидных случаях можно опускать индекс 'T'.

Переводом, определяемым схемой T, или □(т) называют множество пар цепочек $\{(x, y) \mid (S, S) \Rightarrow^* (x, y), x \in \sum^* \text{ и } y \in \Delta^*\}$.

Пример 5.2

Пусть СУ-схема задана следующим образом f31:

T = ({S}, {i, +}, {i, +}, R, S),

где множество правил R содержит правила

(1) $S \to + S(1)\ S(2), S(1) + S(2)$

(2) $S \to i, i$

Рассмотрим вывод в данной СУ-схеме:

$(S,S) \Rightarrow_{(1)} (+S^{(1)}S^{(2)},$

$S^{(1)} + S^{(2)}) \Rightarrow_{(1)} (+ + S^{(3)}\ S^{(4)}\ S^{(2)},$

$S^{(3)} + S^{(4)} + S^{(2)}) \Rightarrow_{(1)} (+ + + S^{(5)}\ S^{(6)}\ S^{(4)}\ S^{(2)},$

$S^{(5)} + S^{(6)} + S^{(4)} + S^{(2)}) \Rightarrow_{(2)} (+ + + a\ S^{(6)}\ S^{(4)}\ S^{(2)},$

$a + S^{(6)}\ S^{(4)} + S^2) \Rightarrow_{(2)} (+ + + a\ b\ S^{(4)}\ S^{(2)} a + b + S^{(4)} + S^{(2)})$

$\Rightarrow_{(2)} (+ + + a\ b\ c\ S^{(2)},$

$+ b + c + S^{(2)}) \Rightarrow_{(2)} (+ + + a\ b\ c\ d, a + b + c + d).$

Входной Цепочке + + + iiii соответствует выходная Цепочка i + i + i + i.

Очевидно, что перевод, определяемый

СУ-схемой, имеет вид:

τ(T) = {(x, i(+ i)k) | k ≥ O и x - префиксная запись выражения i(+i)k}.

Примечание: если T = (T, ∑, Δ, R, S) — СУ-схема, то τ(т) называется синтаксически управляемым переводом.

Примечание: G_i = (N, ∑, P, S), где P = {A → α | A → α, β, ∈ R}, называется входной грамматикой СУ-схемы т.

Примечание: G_0 = (N, Δ, P', S), где P' = {A → β | A → α, β, ∈ R}, называется выходной грамматикой СУ-схемы т.

К достоинствам СУ-схемы относится не только ее простота, но и нагладь-носить ее интерпретации. Синтаксически управляемый перевод можно трактовать как метод преобразования деревьев выводов входной грамматики G_i в деревья выводов выходной грамматики G_0. Перевод данной входной цепочки и x можно получить, построив ее дерево вывода, затем преобразовав это дерево в дерево вывода в выходной грамматике и, наконец, взяв крону выходного дерева в качестве перевода цепочки x[3].

Алгоритм 5.1 Преобразование деревьев при помощи СУ-схемы

Вход: СУ-схема T = (N, ∑, Δ, R, S) с входной грамматикой G_i = (N, ∑, P, S) и выходной грамматикой G_0 = (N, Δ, P', S) и дерево вывода D в G_i, с кроной, принадлежавшей ∑*.

Выход: Некоторое дерево вывода D' в G_0, такое, что если x и y - кроны деревьев D и D' соответственно, то (x, y) ∈ τ(T)

Описание алгоритма:

1.Пусть Данный шаг применяется к внутренней вершине n Дерева D, имею шей k прямых потомков $n_1, \ldots, n_k$.

1.1 Устранить из множества вершин $n_1, \ldots, n_k$ листья, помеченные терминальными символами или ε.

1.2 Пусть A → α — правило входной грамматики G_i, соответствую шее вершине n и ее прямым потомкам, т. е. A — метка вершины п, и α образуется конкатенацией меток вершин $n_1, \ldots, n_k$. Выбрать из R некоторое правило вида A → α, β (если таких правил несколько, то выбор произволен).

Переставить оставшиеся прямые потомки вершины n (если они есть) согласно соответствию между вхождениями не терминалов в α и β. (Поддеревья, корнями которых служат эти потомки, переставляются вместе с ними).

1.3 Добавить в качестве прямых потомков вершины п листья с метками так, чтобы метки всех ее прямых потомков образовали цепочку β.

1.4 Применить Шаг 1 к прямым потомкам вершины п, не являвшимся листьями, в порядке слева направо.

2. Повторять Шаг 1 рекурсивно, начиная с корня дерева D.

Результирующим деревом будет D' .

Пример 5 3

Рассмотрим СУ-схему 1 = ({S, A}, {0, 1}, {a, Ь}, R, S), где R состоит из

правил:

(1)S → 0AS, SAa

(2)A → 0SA, ASa

(3)S → 1, b

(4) A → 1, b

Во входной грамматике этой схемы цепочка 0010111 имеет вывод 1232343 (дерево вывода приведено на рис. 5.1).

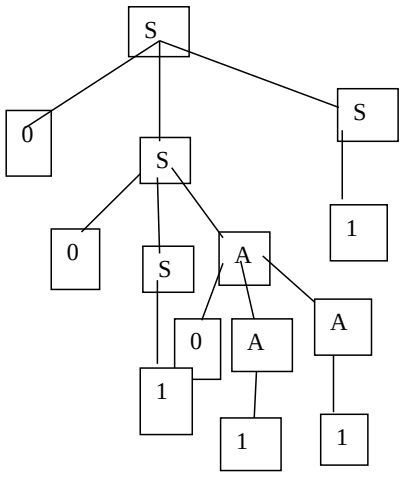


Рис. 5.1. Дерево вывода цепочки 0010111

Преобразуем это дерево при помощи алгоритма 5.1.

Шаг 1. Применим шаг 1 алгоритма к корню Дерева S:

Устраняем левый лист дерева, помеченный 0. Находим правило входной грамматики (1) S → 0AS, соответствующее корню дерева S. У этого правила только один элемент перевода SA, который определяет, что нужно поменять местами прямые потомки корня A и S. Добавляем прямой потомок к корню дерева и помечаем его a.

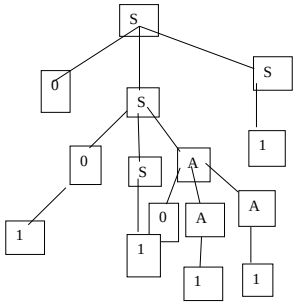
результат этих действий приведена на рис. 5.2, а.

Шаг 1. Рассматриваем прямой потомок корня дерева – вершину, помеченную символом A.

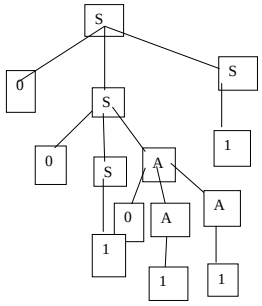
Устраняем лист 0. Используя правило (2) СУ-схемы, меняем местами прямые потомки рассматриваемой вершины. Добавляем прямой потомка к рассматриваемой вершине и получаем дерево, приведённое на рис. 5.2, б.

Шаг 1. Рассматриваем прямой потомок корня дерева – вершину, помеченную символом S.

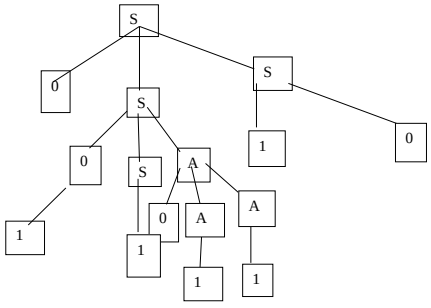
Устраняем лист 1. Находим соответствующее правило СУ-схемы – правило номер (3), и так как потомков у рассматриваемой вершины нет, то создаём прямой потомок b Для рассматриваемой вершины и получаем Дерево, приведённое на рис. 5.2, в


S
0
S
S
0
S
A
1
0
A
A
1
1
1
1

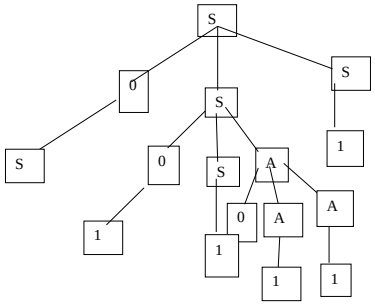
а


S
0
S
S
0
S
A
1
0
A
A
1
1
1

б


S
0
S
S
0
S
A
1
0
1
0
A
A
1
1
1

в


S
0
S
S
S
0
S
A
1
0
A
A
1
1
1

г

Рис. 5.2. Преобразование деревьев

Продолжая рекурсивно выполнение шага 1 алгоритма, получаем результирующее дерево, приведённое на рис. 5.2, г.

В [3] Доказаны следующие утверждения:

1. Если x и y – кроны деревьев D и D’ из алгоритма 5.1, то (x, y) ∈ τ(T).
2. Если (x, y) ∈ τ (T), то существуют Дерево вывода D с кроной x и такая последовательность выборов вершин при обращении к Шагу 2.2 алгоритма, что в результате получается Дерево D □ с кроной y.

Замечание: порядок применений шага 2 алгоритма 5.1 к вершинам дерева не важен. Можно выбрать любой порядок, при котором каждая внутренняя вершина рассматривается точно один раз.

Примечание: СУ-схема T = (N, ∑, Δ, R, S) называется простой, если для каждого правила A → α, β ∈ R соответствующие друг другу вхождения не терминалов встречаются в α и β в одном и том же порядке.

Примечание: перевод, определяемый простой СУ-схемой, называется простым синтаксически управляемым переводом.

Соответствие нетерминалов в выводимой паре простой СУ-схемы определяется порядком, в котором эти нетерминалы появляются в цепочках, поэтому для нее легко построить транслятор, представляющий собой преобразователь с магазинной памятью.

По этой причине простые СУ-переводы образуют важный класс переводов.

Пример 5 .4

Рассмотрим простую СУ-схему, имеющую следующие правила:

(1)E → (E), E

(2)E → E + E , E + E (6)T → i , i

(3)E → T, T (7)A → (E + E) , (E + E)

(4)T → (T), T (8) A → T, T

(5)T → A * A, A * A

и вывод входной цепочки ((i * (i * i) + i) * i), равный 3457358684586863686.

Соответствующий вывод пар цепочек имеет вид:

(E, E) $\Rightarrow_{(3)}$ (T, T) $\Rightarrow_{(4)}$ ((T), T) $\Rightarrow_{(5)}$ ((A * A), A * A)

$\Rightarrow_{(7)}$ (((E + E) * A, (E + E) * A)$\Rightarrow_{(3)}$ (((T + E) * A, (T + E) * A)

$\Rightarrow_{(5)}$ (((A * A + E) * A, (A * A + E) * A)$\Rightarrow_{(8)}$ (((T * A + E) * A, (T * A + E) * A)$\Rightarrow_{(6)}$ (((i * A + E) * A, (i * A + E) * A) $\Rightarrow_{(8)}$ (((i * T + E) * A, (i * T + E) * A) $\Rightarrow_{(4)}$ (((i * (T) + E) * A, (i * T + E) * A)$\Rightarrow_{(5)}$ (((i * (A * A) + E) * A, (i * A * A + E) * A) $\Rightarrow_{(8)}$ (((i * (T * A) + E) * A, (i * T * A + E) * A) $\Rightarrow_{(6)}$ (((i * (i * A) + E) * A, i * i * A + E) * A) $\Rightarrow_{(8)}$ (((i * (i * T) + E) * A , (i * i * T + E) * A) $\Rightarrow_{(6)}$ (((i * (i * i) + E) * A ,(i * i * i + E) * A) $\Rightarrow_{(3)}$ (((i * (i * i) + T) * A, (i * i * i + T) * A) $\Rightarrow_{(6)}$ (((i * (i * i) + i) * A, (i * i * i + i) * A) $\Rightarrow_{(8)}$ (((i * (i * i) + i) * T, (i * i * i + i) * T) $\Rightarrow_{(6)}$ (((i * (i * i) + i) * i, (i * i * i + i) * i).

Для входной цепочки ((i * (i * i) + i) * i) СУ-схема порождает выходную цепочку (i * i * i + i) * i. анализ позволяет сделать вывод, что СУ-схема отображает арифметические выражения, порождаемые грамматикой G_0, в арифметические выражения, не содержащие избыточных скобок.

Рассмотрение СУ-схем позволяет понять идеи, лежащие в основе методов описания синтаксически управляемого перевода: включение в грамматику, описывающую порождение цепочек языка, элементов перевода, позволяющие описывать порождение выходных Цепочек.

Для получения дополнительных сведений о СУ-схемах можно обратиться к [3, 4].

5.3. Транслирующие грамматики

Рассматривая процесс перевода инфиксных арифметических выражений в польскую инверсную запись, попытаемся построить некоторый процессор, выполняющий этот перевод.

Если на входной ленте процессора находится цепочка a + b * c, то процессор должен работать следующим образом:

- □ прочитать входной символ, a;
- □ выдать символ, a на выходную ленту;
- □ почитать входной символ '+';
- □ почитать входной символ Ь;
- □ выдать символ Ь на выходную ленту;
- □ почитать входной символ '*';
- □ почитать входной символ c;
- □ выдать символ c на выходную ленту;
- □ выдать символ '*' на выходную ленту;
- □ выдать символ '+' на выходную ленту.

Этот план работы определяется тем, что порядок следования операндов в инфиксной записи и ПОЛИЗ совпадает, а операции в ПОЛИЗ должны следовать сразу за своими операндами. Если операцию Чтения символа из входной ленты мы будем обозначать Читаемыми символами, а операцию записи – символами, помещаемыми на выходную ленту, заключёнными в фигурные скобки (операционными символами), то работа по переводу может быть записана следующим образом:

a {a} + b {b} * c {c} {*} {+}.

Транслирующая грамматика – это КС-грамматика, множество терминалов которой разбито на два множества: множество входных и множество операционных символов.

Дадим формальное определение транслирующей грамматики.

Определение: транслирующей грамматикой называется пятерка объектов $G^T = (N, \sum i, \sum a, P, S)$, где — словарь входных символов, $\sum a$ — словарь операционных символов, N — нетерминальный словарь, $S \in N$ — начальный символ транслирующей грамматики, P — конечное множество правил вывода вида $A \rightarrow \alpha$, в которых $A \in N$, а $\alpha \in (\sum l \cup \sum a \cup N)^*$.

Транслирующая грамматика $G_0{}^T$, описывающая перевод инфиксных арифметических выражений в ПОЛИЗ, содержит следующие правила:

(1) E → E + T {+} (4) T → P

(2) E → T

(5) P → i {i}

(3) T → T * P {*} (6) P → (E)

Грамматика, полученная из транслирующей грамматики путем вычеркивания всех операционных символов, называется входной грамматикой для этой транслирующей грамматики. Язык, порождаемый входной грамматикой, называется входным языком.

Например, входной грамматикой G0 для транслирующей грамматики $G0^T$ является КС-грамматика для инфиксных арифметических выражений, содержащая правила:

(1) E → E + T (4) T → P

(2) E → T (5)

P → i

(3) T → T * P (6)

P → (E)

Рассмотрим левый вывод цепочки i + i * i во входной грамматике G_0:

E ⟹ E + T ⟹ T + T ⟹ P + T ⟹ i + T ⟹ i + T * P ⟹ i + P * P ⟹

i + i * P ⟹ i + i * i.

Применив эту же последовательность правил к соответствующим нетерменалам транслирующей грамматики $G_0{}^T$ получим следующий левый вывод:

E ⟹ E + T {+} ⟹ T + T {+} ⟹P + T {+} ⟹ i {i} + T {+} ⟹i {i} + T * P {*} {+} ⟹ i {i} + P * P {*} {+} ⟹ i {i} + i {i} * P {*} {+} ⟹ i {i} + i {i} * i {i} {*} {+}.

Цепочки языка, порождаемого транслирующей Грамматикой, называют активными цепочками. Входной частью активной цепочки называется последовательностью входных символов, полученная из активной цепочки после удаления всех операционных символов. Операционной частью активной цепочки называется последовательность операционных символов, полученная путем вычёркивания из активной цепочки всех входных символов.

Например, последовательность i + i * i является входной Частью активной цепочки i {i} + i {i} * i {i} {*} {+}, а последовательностью {i}{i}{i}{*}{+} – ее операционной частью.

Примечание: множество всех пар, первыми элементами которых являются входные части активной цепочки, а вторыми элементами — операционные части активной цепочки, называется синтаксически управляемым переводом τ(G^T), определяемым транслирующей грамматикой G^T, .

т

В восходящих методах обработки языков широко применяются постфиксные транслирующие грамматики.

Примечание: транслирующая грамматика называется постфиксной транслирующей грамматикой тогда и только тогда, когда все операционные символы в правых частях правил вывода расположены правее всех входных и нетерминальных символов.

Примером постфиксной транслирующей Грамматики является Грамматика $\boldsymbol{G_0}^{\boldsymbol{T}}$.

Для получения транслирующей грамматики входной язык описывается входной КС-грамматикой. Затем в правила вывода этой грамматики вставляются операционные символы для описания семантических действий, связанных с соответствующими правилами.

Рассмотрим получение транслирующей грамматики $G0^T$ по входной Грамматике G0.

В ПОЛИЗ операнды располагаются в той же последовательности, что и в инфиксной записи, а знаки операций следуют непосредственно после своих операндов в том порядке, в котором они выполняются. Для того чтобы идентификатор i выдавался сразу после его прочтения, правило (6) Грамматики GJ преобразуется к виду P → i {i}. Чтобы знак операции сложения выдавался непосредственно после своих операндов, правило (1) заменяется на E → E + T {+}. Это правило интерпретируется следующим образом: обработка арифметического выражения E состоит из обработки первого операнда E, чтения символа '+', обработки второго операнда T и выдачи символа '+'. Аналогичные рассуждения позволяют следующим образом преобразовать правило (3) грамматики:

T → T * P {*}.Один и тот же перевод можно определить разными транслирующими грамматиками

Пример 5.5

Пусть две транслирующие грамматики $G_1{}^T$ и $G_2{}^T$, определяющие перевод в ПОЛИЗ инфиксных бесскобочных арифметических выражений, выполняемых в порядке написания операций, выглядят следующим образом:

$G_1{}^T$

(1)E → E + T {+}

(2)E → E * T {*}

(3)E → T

(4)T → i {i}

$G_2{}^T$

(1)E → i {i} R

(2)R → + i {i} {+} R

(3)R → * i {i} {*} R

(4)R → ε

Рассмотрим выполнение перевода цепочки i + i * i с использованием транслирующих грамматик $G_1{}^T$ и $G_2{}^T$.

При использовании Грамматики $G_1{}^T$ активная цепочка порождается следующим образом:

E $\Rightarrow_{(2)}$ (2) E * T {*}
$\Rightarrow_{(1)}$ E + T {+} * T {*}
$\Rightarrow_{(3)}$ T + T {+} * T {*}
$\Rightarrow_{(4)}$ i {i} + T {+} * T {*}
$\Rightarrow_{(4)}$i {i} + i {i}{+} * T {*}
$\Rightarrow_{(4)}$ i {i} + i {i}{+} * i {i}{*}

Полученная операционная часть цепочки {i}{i}{+}{i}{*} является переводом входной цепочки i + i * i.

Использование второй Грамматики $G_2{}^T$ позволяет получить следующе порождение активной Цепочки:

E $\Rightarrow_{(1)}$ i {i} R
$\Rightarrow_{(2)}$ i {i} + i {i}{+} R
$\Rightarrow_{(3)}$ i {i} + i {i}{+} * i {i}{*} R
$\Rightarrow_{(4)}$ i {i} + i {i}{+} * i {i}{*}

Полученная операционная часть цепочки {i}{i}{+}{i}{*} также является переводом входной цепочки. На рис. 5.3 приведены деревья вывода Цепочки i{i} + i {i}{+} * i {i}{*} с использованием грамматик $G_1{}^T$ и $G_2{}^T$ соответственно.

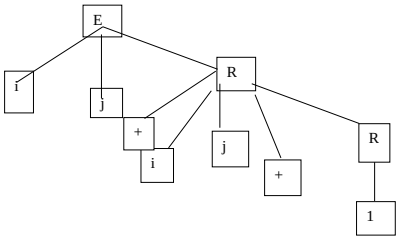


Рис. 5.3. Деревья вывода цепочки i {i} + i {i}{+} * i {i}{*} в грамматиках $G_1{}^T$

Во многих практических приложениях появление входного символа в активной цепочке можно интерпретировать как обозначение операции чтения этого символа некоторым устройством, а вхождению операционного символа в правило вывода можно сопоставить операцию выдачи символа, заключённого в фигурные скобки. При таком подходе входную часть активной цепочки называют входной цепочкой, операционную часть – входной Цепочкой, а транслирующую грамматику, определяющую перевод, грамматикой цепочечного перевода.

Примером грамматики цепочечного перевода является грамматика $G_0{}^T$:

(1) E → E + T {+}
(2) E → T
(4) T → P
(3) T → T * P {*}

(5)P → i {i}
(6)P → (E)

Другая возможная интерпретация операционных символов - имена семантических процедур, вызываемых при обработке входной цепочки. В этом случае активная цепочка определяет последовательностью вызовов семантических процедур и моменты времени, в которые выполняются эти вызовы по отношению к моментам чтения входных символов.

5.4 Атрибутные транслирующие грамматики

При определении транслирующей грамматики в понятие **входного символа** не включалось представление о том, что он является **лексемой** и состоит из двух компонентов: **типа и значения**. На самом деле транслирующая грамматика способна описывать перевод только той части символа, которая задаёт его тип. Рассмотрим, как можно расширить понятие грамматики цепочечного перевода, чтобы использовать при переводе оба компонента входного символа. **Расширенная транслирующая грамматика получила название атриибитной транслирующей грамматики (АТ-Грамматики)**. АТ-грамматики были предложены Дональдом Кнутом и в дальнейшем изучались рядом учёныних. материал данного раздела в основном базируется на результатах, приведённых в [2, 27, 28].

В АТ-Грамматике символы снабжаются **атрибутами**, которые могут принимать значения из некоторого множества допустимых значений и **интерпретируются** как **семантическая** информация, связанная с конкретным вхождением символа в правило вывода грамматики. При таком подходе значения атрибутов связываются с вершинами дерева вывода в АТ-грамматике, а правила вычисления значений атрибутов сопоставляются правилам вывода грамматики.

Все атрибуты нетерминальных и операционных символов делятся на синтезированные и унаследованные атрибуты.

5.4.1 Синтезированные атрибуты

Рассмотрим получение АТ-Грамматики из исходной транслируюwей грамматики.

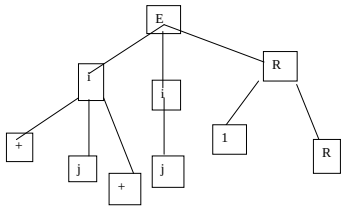
Пример 5.6 Пусть задана транслирующая грамматика $G_3{}^T$, допускающая в качестве входа арифметические выражения, построенные из символов множества

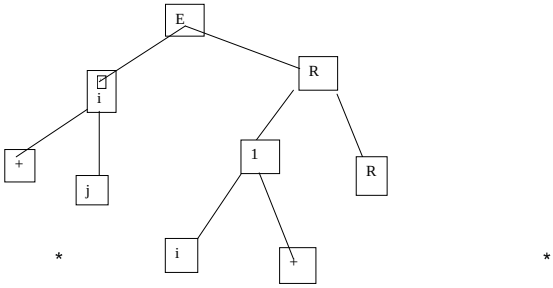
{c, +, *, (,)}, где c - лексема, является целочисленной константой и порождающая на выходе значение этого выражения:

(O)S → E {ОТВЕТ}

(1)E → E + T (2)E → T (3)T → T * P (4)T → P (5)P → c (6)P → (E)

На рис. 5.4, а изображено дерево вывода для входной цепочки $c_5 + c_2 * c_3$ (индексы обозначают значения лексем). Этой входной цепочке должна соответствовать выходная цепочка {ОТВЕТ}.





а **б**

Рис. 5.4. Дерево вывода цепочки $c_5 + c_2 * c_3$ во входной (а) и атрибутной транслирующей (б) грамматиках.

Поскольку любое вхождение не терминалов Е, Т и Р в дерево вывода представляет собой некоторое подвыражение входного выражения, введём для каждого из символов Е, Т и Р по одному атрибуту, значением которого будет значение подвыражения, порождаемого этим не терминалом.

Входной символ с и операционный символ {ОТВЕТ} также имеют по одномы атрибуту. Значение атрибута символа с равно значению константы, которулю он представляет, а значением символа {ОТВЕТ} является результат выражения.

Все атрибуты символов могут принимать любые целочисленные значения из области допустимых для выражений, описываемых соответствующей входной грамматикой G_3. На рис. 5.4, б изображено дерево вывода той же цепочки, что и на рис. 5.4, а, но на этом дереве не терминалы и операционный символ {ОТВЕТ} помечены значениями своих атрибутов.

Выберем для каждого атрибута каждого вхождения символа в правило вывода грамматики уникальное имя и включим атрибуты в правила вывода в виде подстрочных индексов соответствующих символов.

Замечание: одни и те же имена атрибутов можно использовать в нескольких правилах, т. к. они локальны в правиле грамматики.

Сопоставим каждому правилу вывода Грамматики правило вычисления значения атрибута не терминала из левой Части правила, которому соответствует нетерминальная вершина дерева вывода.

Рассмотрим, например, вершину Т дерева, изображённого на рис. 5.4, а. Правило вывода, применённое к этой вершине, имеет вид: Т → Т * Р. Оно определяет, Что значение подвыражения, порождённого не терминалом из левой Части правила, равно значению подвыражений, порождённых символами

P и T из правой Части правила, которые являются прямыми потомка- ми вершины T. Если p - атрибут символа T из левой Части правила, а q и r - атрибуты символов T и P из правой части правила, то правило вычисления значения атрибута p будет иметь вид: p ⟵ q * r, где символ '⟵' - знак операции присваивания.

Таким образом, начиная с атрибутов входных символов и поднимаясь по Дереву от кроны к корню, можно определить все значения атрибутов Нетер- мина лов из левых Частей правил вывода. Правила вычисления значений атрибутов Для АТ-Грамматики $G_3{}^T$, полученной по транслирующей uрамматике $G_3{}^T$, будут выглядеть следующим образом:

(O)S→E_q {OTBET}p

(4)T_p* P_q

q ⟵ p

p ⟵ q

(1) E_p → E_q +T_r

(5) P_p → (E_q)

p ⟵ q + r p ⟵ q

(2) E_p → T_q

(6) P_p → C_q

p ⟵ qp ⟵ q

(3)T_p→T_q * P_r

p ⟵ q * r

Атрибуты, значения которых вычисляются при движении по дереву вывода снизу вверх, традиционно называют синтезированными. Термин "синтезированный" подчёркивает, что значение атрибута синтезируется из значений атрибутов потомков.

Рассмотренные ранее правила вычисления значений атрибутов не распространяются на атрибут операционного символа {OTBET}, который относится к классу унаследованных атрибутов.

5.4.2. **Унаследованные атрибуты**

Термин "унаследованный" означает то, что значение атрибута зависит от значений атрибутов предка символа или атрибутов его соседей в дереве вывода. Например, в грамматике G3T, значение атрибута r символа {OTBET} равно значению атрибута не терминала E (соседа слева), порождающего все выражение. Оно может быть вычислено только после того, как будут определены значения всех синтезированных атрибутов не терминалов.

Рассмотрим ещё один пример АТ-Грамматики.

Пример 5.7

Пусть входная КС-грамматика G, порождающая описания переменных в некотором языке программирования, выглядит следующим образом:

1)D → t i L 2)L → , i L 3)L → ε

Множество входных символов этой грамматики состоит из лексем: запятая, идентификатор i и имя типа t.

Семантика обработки описания заключается в занесении типа переменной в определённое поле элемента таблицы идентификаторов. Эту операцию можно выполнить с помощью семантической процедуры УСТАНОВИТЬ_ТИП с двумя параметрами: указатель на элемент таблицы индефикаторов, соответствующей описываемой переменной, и тип переменной. Процедуру УСТАНОВИТЬ_ТИП лучше всего вызывать сразу после распознавания идентификатора. Указанная последовательность действий может быть описана транслирующей грамматикой G_T:

D → t i {ТИП} L

L → , i {ТИП} L

L → ε

В этой грамматике вызову процедуры УСТАНОВИТЬ_ТИП соответствует операционный символ {ТИП}.

Введём в транслирующую грамматику G^T атрибуты и правила их вычисления. Входные символы t и i имеют по одному атрибуту. Значением атрибута символа i является указатель на элемент таблицы идентификаторов, а атрибут символа t может принимать значения из множества {ЦЕЛЫЙ, ВЕЩЕСТВЕННЫЙ, ЛОГИЧЕСКИЙ}.

Операционный символ {ТИП} должен иметь два унаследованных атрибута, значения которых совпадают со значениями соответствующих фактических параметров процедуры УСТАНОВИТЬ_ТИП. Значения унаследованных атрибутов символа {ТИП} Для первого правила вывода приравниваются значениям соответствующих атрибутов входных символов i и t, входящих в правую часть того же правила левее символа {ТИП}.

Во второе правило тип описываемых переменных можно передать через унаследованный атрибут не терминала L. Значение этого унаследованного атрибута будет передаваться по дереву сверху вниз, начиная с вершины, где он получает начальное значение, равное значению атрибута входного символа t.

Атрибутная транслирующая Грамматика G^A выглядит следующим образом:

D → $t_r\ i_a$ {ТИП}$_{a1,\ r1}$ L_{r2}

$a_1 \leftarrow a$

$r_{1,}\ r_2 \leftarrow r$

(1)

$L_r \rightarrow , i_a$ {ТИП}$_{a_1,\ r_1}$ L_{r_1}

$a_1 \leftarrow a$

$r_{1,}\ r_2 \leftarrow r$

(2)

$L_r \rightarrow \varepsilon$

Замечание: запись атрибутного правила в виде $r_1, r_2 \longleftarrow$ r означает, что значение r присваивается одновременно атрибутам r_1 и r_2.

Входной цепочке $t_{\text{ЦЕЛЫЙ}}$ i_5, i_9, i_2 соответствует дерево вывода в грамматике G^{A}, изображённое на рис. 5.5.

,

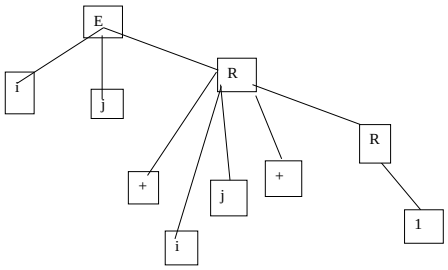


Рис. 5.5. Дерево вывода

В рассмотренных примерах все операционные символы имели унаследованные атрибуты. При определении атрибутной транслирующей грамматики можно обойтись без синтезированных атрибутов операционных символов. Необходимость в синтезированных атрибутах операционных символов может возникнуть в некоторых практических реализациях переводов, поэтому в определение АТ-Грамматики включены оба типа атрибутов.

5.4.3. Определение атрибутной транслирующей грамматики

Атрибутная транслирующая грамматика - это транслирующая грамматика, обладающая следующими дополнительными свойствами:

1.Каждый символ грамматики (входной, нетерминальный и операционной) имеет конечное множество атрибутов, и каждый атрибут имеет (возможно, бесконечное) множество допустимых значений.

2.Все атрибуты нетерминальных и операционных символов делятся на синтезированные и унаследованные.

3.Унаследованные атрибуты подчиняются следующим правилам:

• каждому вхождению унаследованного атрибута в правую часть правела вывода сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов символов, входящих в левую или правую часть данного правила;

• для каждого унаследованного атрибута начального символа грамматики задаётся начальное значение.

4.Правила вычисления значений синтезированных атрибутов определяются следующим образом:

• каждому вхождению синтезированного атрибута нетерминального символа в левую Часть правила вывода сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов символов, входящих в левую или правую часть данного правила;

• каждому синтезированному атрибуту операционного символа сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов данного символа.

• каждому синтезированному атрибуту операционного символа сопоставляется правило, позволяющее вычислить значение этого атрибута как функцию некоторых других атрибутов данного символа.

Будем записывать атрибуты в виде индексов соответствующих символов АТ-грамматики, при этом Для каждого атрибута будем указывать, к какому классу этот атрибут относится. Например:

$X_{a,b}$

синтезированный α

унаследованный b

Правила вычисления атрибутов будем записывать в виде операторов присваивания, левая часть которых - атрибут или список атрибутов, а правая часть - функция.

Рассмотрим АТ-грамматику G_A , описывающую перевод оператора присваивания некоторого языка программирования в цепочку тетрад с кодами операций: СЛОЖИТЬ, УМНОЖИТЬ, ПРИСВОИТЬ. Левой частью оператора присваивания является идентификатор, а правой частью - бесскобочное арифметическое выражение, выполняемое слева направо в порядке написания операций.

Входными символами грамматики являются символы множества {i, +, *, =}, где i - лексема, представляющая идентификатор. Каждый идентификатор имеет один атрибут, значением которого является указатель на соответствующий элемент таблицы идентификаторов.

Операционные символы {+}, {*}, {:=} соответствуют кодам операций тетрад, получаемых на выходе. Символы {+} и {*} имеют по три унаследованных атрибута, значениями которых являются указатели на элементы таблицы, в которой хранятся значения левого операнда, правого операнда и результата соответственно. Операционный символ {:=} имеет два унаследованных атрибута. Значение первого атрибута - указатель на элемент таблицы идентификаторов, соответствующий идентификатору из левой Части оператора присваивания, а значением второго атрибута является указатель на элемент таблицы, в котором хранится значение выражения из правой Части оператора присваивания.

Словарь нетерминальных символов состоит из символов S, E и R. Начальный символ S не имеет атрибутов, символ E имеет один синтезированный атрибут, значением которого является указатель на элемент таблицы, содержащий значение выражения, порождаемого E, а символ R имеет два атрибута: унаследованный (первый) и синтезированный. Значением унаследованного атрибута является указатель на элемент таблицы, соответствующий промежуточному результату, предшествующему R. Этот промежуточный результат является также левым операндом первой операции, порождаемой

нетерминальным символом R, если такая операция имеет место. Значение синтезированного атрибута символа R - это указатель на элемент таблицы, соответствующий значению подвыражения, которое получается после присоединения цепочки, порождённой символом R, к цепочке, представляющей левый операнд.

Правила вывода АТ-Грамматики G_A с соответствующими правилами вычисления следующих атрибутов:

E_t

синтезированный t

$R_{p,t}$

унаследованный p

синтезированный t

Атрибуты операционных символов унаследованные.

(1) S→i_{p_1} := E_{q_1} {:=}p_2 , q_2

$p_2 \leftarrow p_1$

$q_2 \leftarrow q_1$

(2)E_{t_2}→i_{p_1} R_{p_2} , t_1

$p_2 \leftarrow p_1$

$t_2 \leftarrow t_1$

(3)$R_{p_1,\ t_2}$ →
i_{q_1} {+}p_2, q_2, r_1 $R_{p_2,\ t_1}$

$r_1, r_2 \leftarrow$ GETNEW

$p_2 \leftarrow p_1$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

(4)$R_{p_1,\ t_2}$→* i_{q_1} {*}p_2, q_2, r_1 $R_{p_2,\ t_1}$

$r_1, r_2 \leftarrow$ GETNEW

$p_2 \leftarrow p_1$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

(5)$R_{p_1,\ t_2}$→ε

$p_2 \leftarrow p_1$

В атрибутных правилах, связанных с правилами вывода (3) и (4), используется процедура-функция без параметров GETNEW, которая выдаёт значение указателя на свободную позицию таблицы, где будут запоминаться промежуточные результаты. Строго говоря, эта процедура не является функцией, т. к. при разных обращениях к ней получаются разные результаты. Таким образом, пользуясь процедурой GETNEW, мы несколько отступаем от определения АТ-грамматики. Вместо того чтобы использовать процедуру GETNEW, можно ввести дополнительные атрибуты для запоминания адресов занятых позиций таблицы. Однако при использовании процедуры GETNEW

атрибутные правила получаются более простыми, и такой подход можно порекомендовать при практическом конструировании компиляторов.

АТ-Грамматики используются для получения атрибутных деревьев вывода, атрибутных активных цепочек и атрибутных переводов.

Процедура построения атрибутного дерева вывода включает следующие действия:

1. По соответствующей транслирующей грамматике построить дерево вывода активной цепочки, состоящей из входных и операционных символов без атрибутов.
2. Присвоить начальные значения унаследованным атрибутам начального символа грамматики.
3. Присвоить значения атрибутам входных символов, входящих в дерево вывода.
4. Найти атрибут, отсутствующий в дереве вывода, аргументы правила вычисления которого уже известны. Вычислить значение этого атрибута и добавить его в дерево.

Повторять шаг 4 до тех пор, пока значения всех атрибутов символов, входящих в дерево вывода, не будут вычислены или пока шаг 4 может выполняться.

На рис. 5.6 приведено атрибутное дерево вывода в АТ-грамматике, соответствующее входной цепочке $i_5 = i_7 + i_2 * i_3$ при условии, что процедура GETNEW выдаёте последовательные адреса ячеек, начиная с адреса 100. Заметим, что при применении правила (5) синтезированному атрибуту не терминала R присваивается значение его унаследованного атрибута. Затем это значение передаётся вверх по дереву вывода как синтезированный атрибут не терминала из левой части правил (4), (3) и (2) и используется в качестве второго атрибута операционного символа {ПРИСВОИТЬ}.

Последовательность атрибутных входных и операционных символов, полученная по атрибутному дереву вывода в АТ-грамматике, называется атрибутной активной цепочкой.

Множество пар, первым элементом которых является атрибутная входная цепочка, а вторым элементом — атрибутная последовательность операционных символов атрибутной активной цепочки, называется атрибутным

Пример 5.8

Грамматика GA для входной Цепочки $i_5 = i_7 + i_2 * i_3$ определяет следующую атрибутную последовательность операционных символов:

$\{+\}_{7,\ 2,\ 100}\ \{*\}_{100,\ 3,\ 101}\ \{:=\}_{5,\ 101}$.

Определение: дерево вывода в АТ-грамматике называется завершённым, если в результате применения процедуры построения дерева значения всех атрибутов всех символов окажутся вычисленными.

Хотя для каждого атрибута, входящего в Дерево вывода, существует правило его вычисления, может оказаться, что между атрибутами возникла круговая зависимость, в результате чего нельзя получить завершённое дерево.

Определение: АТ-грамматика называется корректной тогда и только тогда, когда в результате применения описанной процедуры построения атрибутного дерева вы- вода получается завершённое дерево.

Далее будут рассмотрены два подкласса корректных АТ-Грамматик, которые часто используются при проектировании компиляторов: L-атрибутные транслирующие грамматики и S-атрибутные транслирующие Грамматики.

5.4.4. Вычисление значений атрибутов

Вернёмся ещё раз к проблеме вычисления значений атрибутов. После построения дерева вывода входной цепочки возникает вопрос о порядке вычисления значений атрибутов. По определению, атрибут можно вычислить, если известны значения всех атрибутов, от которых зависит его значение. Число деревьев вывода так же, как и входных цепочек, бесконечно, поэтому важно по самой грамматике уметь определять, является ли множество её правил вычисления атрибутов корректным.

Алгоритмы проверки корректности АТ-Грамматики приведены в [2, 25]. Они основаны на построении графа зависимостей и euo анализе.

Узлами графа зависимостей служат атрибуты, которые нужно вычислить, а дугам ставятся в соответствие зависимости, определяющие, какие атрибуты вычисляются раньше, а какие позже.

Граф зависимостей R(D) представляет собой ориентированный Граф, который строится для некоторого дерева вывода D следvующим образом:

□ у злами R(D) являются пары (X, a), где X - узел дерева D, а *a* - атрибут символа, служащего меткой узла X;

□ дуга из узла (X_1, a_1) в узел (X_2, a_2) проводится, если семантическое правило, вычисляющее значение атрибута a_1, непосредственно использует значение атрибута a_1.

Вычисление значения двоичного числа по его символьному представлению можно задать при помощи следующей атрибутной грамматики [24]:

$N_{v_4} \rightarrow L_{v_2,l_2,S_2} . \ L_{v_3,l_3,S_{23}}$

$L_{v_2,l_2,S_2} \rightarrow \ B_{v_1,S_1}$

$V_4 \leftarrow V_2 + V_3 \ V_2 \leftarrow V_1$

$S_2 \leftarrow 0 S_1 \leftarrow S_2$

$S_3 \ \leftarrow -l_3 l_2 \leftarrow 1$

$N_{v_4} \rightarrow L_{v_2,l_2,S_2}$

$B_{v_1,S_1} \rightarrow 0$

$V_4 \leftarrow V_2 V_1 \leftarrow 0$

$S_2 \leftarrow 0$

$L_{v_2,l_2,S_2} \rightarrow L_{v_3,l_3,S_3}\ B_{v_1,S_1}$

$B_{v_1,S_1} \rightarrow 1$

$V_2 \leftarrow V_3 + V_1 V_1 \leftarrow 1$

$S_1 \leftarrow S_2$

$S_3 \leftarrow S_2 + 1$

$l_2 \leftarrow l_3 + 1$

В этой грамматике атрибуты имеют следующую семантику: v – значение (рациональное число), s – масштаб (целое число) и - длина числа (целое число).

Используя грамматику, построим дерево D вывода цепочки 101.01

Для построения узлов графа зависимостей необходимо последовательно рассмотреть вершины дерева D и для каждой из них построить столько узлов, сколько атрибутов имеет символ, которым помечена эта вершина.

Например, для корня дерева, обозначенного символом N с одним атрибуттом, нужно в граф включить один узел, поместив его парой (N, V_4).

Для определения дуг графа необходимо использовать семантические правила. Например, рассмотрим узел Дерева зависимостей (N, V_4). При построении дерева D непосредственные потомки корня N определялись правилом грамматики $N_{v_4} \rightarrow L_{v_2,l_2,S_2}\ .\ L_{v_3,l_3,S_3}$. Атрибут V_4 не терминала N зависит, в соответствии с семантическим правилом $V_4 \leftarrow V_2 + V_3$, от атрибута V_2 левого сына, помеченного символом L, и атрибута V_3 правого сына. Поэтому в граф зависимостей необходимо включить дуги ((L, V_2), (N, V_4)) и ((L, V_3), (N, V_4)). Поступая аналогичным образом, мы можем построить граф зависимостей.

В [24] Доказано, что атрибутная грамматика корректна, если граф зависимостей не имеет циклов.

Общие алгоритмы вычисления атрибутов весьма неэффективны, поэтому на практике используют методы, в которых для определения порядка вычисления не рассматриваются семантические правила. Например, если порядок вычисления атрибутов задаётся методом синтаксического анализа, то он не требует анализа семантических правил и построения графа зависимости.

5.5. Методика разработки описания перевода

При проектировании перевода следует руководствоваться следующими общими соображениями:

1. Описание перевода строится последовательно для конструкций входного языка в порядке их усложнения, начиная с простейших конструкций, например: выражение, оператор присваивания, оператор Цикла.

2.Выходная цепочка (результат перевода) включает в себя объекты (сущности) трех видов:

• объекты, передаваемые в выходную цепочку из входной цепочки;

•объекты, не изменяющиеся в процессе перевода (терминалы выходного языка);

•объекты, генерируемые во время перевода.

3. Транслирующая грамматика определяет порядок применения операционных символов (элементов перевода), строящих выходную цепочку, атрибуты же используются только для передачи значений символов из одних правил в другие.

4. Операционный символ, включаемый в правило вывода грамматики, может генерировать выходную цепочку до тех пор, пока в неё не должен быть включён объект, перемещаемый из входной цепочки и недоступный в данном правиле грамматики.

Проектирование описания перевода некоторой (одной!) конструкции входного языка выполняется в несколько этапов.

Неформальное описание перевода. Неформальное описание перевода представляет собой пару цепочек, первая из которых является конструкцией входного языка, а вторая - ее представлением в выходном языке. При разработке описания перевода рекомендуется:

- строить описание перевода на основе описания синтаксиса и семантики входного языка, учитывая все возможные форматы представления входной конструкции (например, if-then и if-then-else);
- конструкции, перевод которых уже описан и которые являются частью данной конструкции, считать терминальными;
- после построения неформального описания перевода выделить в нем символы, передаваемые из входной цепочки, и символы, генерируемые при его построении.

Описание синтаксиса. Описание синтаксиса выполняется в БНФ или расширенной БНФ. После стандартных преобразований БНФ преобразуется в КС-Грамматику, описывающую рассматриваемую конструкцию входного языка.

Определение транслирующей грамматики. При определении транслирующей грамматики требуется анализировать последовательность используемых при построении перевода правил грамматики, а также моделировать действия, выполняемые операционными символами грамматики. В связи с этим на данном этапе должны быть уточнены:

метод синтаксического анализа, используемый при анализе и переводе Данной конструкции, и способ его реализации;

интерпретация операционных символов (Цепочка символов, помещаемых на выходную ленту, или имя подпрограммы, которую необходимо выполнить).

- Исходными данными при построении транслирующей грамматики являются: КС-Грамматика, описывающая синтаксис конструкции;
- последовательность правил грамматики (разбор), используемых процессором при построении перевода;
- объекты, доступные процессору, выполняющему перевод (для процессора с магазинной памятью это текущий символ входной цепочки и, в

зависимости от метода синтаксического анализа, весь магазин или его Часть).

Для построения транслирующей грамматики необходимо выполнить следующие действия:

1.Выбрать простейшую входную цепочку, соответствующую данной конструкции, и определить ее разбор.

2.В неформальном описании перевода выделить очередной (вначале первый) символ выходной цепочки, который должен быть передан в неё из входной цепочки.

3.На основании доступности (видимости) данных процессору и с учётом разбора, определить правило грамматики, в которое нужно включить элемент перевода (операционный символ), переносящий нужный символ в выходную цепочку. Если действия успешны, то перейти к шагу 5, иначе выполнить следующий шаг 4.

4.Если нельзя выполнить шаг 3, то соответствующий операционный символ включить в наиболее подходящее правило грамматики (в дальнейшем этот символ будет передан в него при помощи атрибутов), определив, если нужно, дополнительную выходную ленту процессора.

5.Возложить на новый (включённый) операционный символ действия по записи в выходную цепочку всех символов, вплоть до следующего символа, который должен быть передан из входной цепочки.

6.Повторить шаги 2-5 для всех символов, передаваемых из входной цепочки.

7.Тестировать транслирующую грамматику на более сложных, например, вложенных входных цепочках.

В простых случаях (простая конструкция входного языка, простое ее представление в выходном языке, удачные метод синтаксического анализа и его реализация, удачное расположение операционных символов) может оказаться, что для описания перевода достаточно только транслирующей грамматики.

Определение атрибутной транслирующей грамматики. При необходимости передачи данных между узлами синтаксического дерева транслирующую грамматику следует расширить до атрибутной транслирующей грамматики.

По определению, передача значений атрибутов в атрибутных транслирующих грамматиках возможна только в следующих направлениях:

- от символа левой части символам правой части правила вывода;
- от символов правой части правила символу левой части этого же правила вывода;
- между символами правой части правила вывода.

Для реализации передачи значения символа необходимо выполнить следующие действия:

1. Определить источник передаваемого значения (символ грамматики) и приписать ему атрибут.

2. Определить приёмник значения (символ грамматики) и приписать ему атрибут.

3. Учитывая ограничения в передаче атрибутов, определить маршрут передачи значения, для чего удобно пользоваться синтаксическим деревом грамматики.

4. Описать передачу значения атрибута правилами АТ-грамматики, стараясь, чтобы функции вычисления атрибутов были простейшими (копирующие правила).

Тестирование АТ-грамматики. Тестирование АТ-грамматики заключается в моделировании работы построенного по ней процессора, выполняющего перевод. При большом числе тестов и/или большой их длине эта работа может быть выполнена только при использовании соответствующих средств автоматизации. Для простых языковых конструкций, когда длина теста и их число невелики, а тестирование выполняется отдельно для каждой конструкции, начиная с простейших, работа эта не только необходима, но и реально выполнима.

Тестирование АТ-Грамматики позволяет определять два вида ошибок:

- ошибки в структуре выходной цепочки (неверный порядок следования символов в выходной цепочке);
- ошибки в значениях символов выходной цепочки, что связано с ошибками в передаче значений атрибутов.

При тестировании реального описания перевода желательно использовать комплексные тесты, определяющие оба вида ошибок.

5.6. Пример разработки АТ-грамматики

Выполним разработку описания перевода оператора цикла, имеющего следующий формат:

for <параметр> = <начальное значение> to <конечное значение> step <шаг>
<тело цикла> next <параметр>

Для упрощения введём следующие ограничения:

<параметр> - идентификатор целой переменной;

<начальное значение>, <конечное значение>, <шаг> - целые положительные константы;

<тело цикла> - оператор цикла или терминал (не подлежащий переводу оператор);

одирование символов входной и выходной программы не рассматривается.

Оператор цикла, который требуется перевести, имеет вид:

for i = c1 to c2 step c3 <тело Цикла> next i

Допустим, что выходной язык не содержит оператора цикла. Тогда результат перевода (рис. 5.9) можно представить следующей последовательностью операторов:

i := c1; m1: if i > c2 then goto m2; <тело цикла>; i := i + c3; goto m1; m2:

Анализ выполняемого перевода позволяет сделать вывод о том, что выходная цепочка включает в себя объекты (сущности) трех видов:

- Объекты, передаваемые в выходную цепочку из входной цепочки (переменная цикла i, начальное значение c1, конечное значение c2, шаг c3).

- Объекты, не изменяющиеся в процессе перевода (множество терминальных символов выходного языка { ;, :, :=, +, >, if, then, goto }.
- Объекты, генерируемые во время перевода (метки m1 и m2).

Составим БНФ, которая описывает синтаксис заданного оператора цикла, выбирая такую структуру описания, которая позволяет проиллюстрировать практически все проблемы синтеза АТ-грамматик:

(1)<оператор цикла> ::= <заголовок цикла><тело цикла>< конец цикла>

(2)<заголовок цикла> ::= for <начальное значение><конечное значение><шаг>

(3)<начальное значение> ::= <идентификатор> = <константа>

(4)<конечное значение> ::= to <константа>

(5)<шаг> ::= step <константа>

(6)<тело цикла> ::= <оператор цикла>

(7)<тело цикла> ::= <терминал>

(8)<конец Цикла> ::= next <идентификатор >

Введём обозначения: s - <оператор цикла>, A - <заголовок цикла>, B - <тело цикла>, C - < конец цикла >, D - <начальное значение>, E - <конечное значение>, F - <шаг>. Тогда правила вывода КС-грамматики, описывающей синтаксис входного языка, имеют вид:

(1) S → A B C (5) F → step cns

(2) A → for D E F (6) B → s

(3) D → id = cns (7) B → term

(4) E → to cns (8) C → next id

При построении транслирующей грамматики требуется сделать некоторые предположения о ходе выполнения перевода и порядке анализа входной цепочки и построения выходной цепочки.

Для определённости будем считать, что:

- используется восходящий метод синтаксического анализа, выполняемый процессором с магазинной памятью;
- выполняется цепочечный перевод, в котором операционный символ вида {OUT := "str"} означает запись в выходную ленту OUT цепочки символов "str".

Определим действия, выполняемые первым по порядку элементом перевода (операционным символом), и правило, в которое он должен быть помещён. Для этого ещё раз рассмотрим неформальное описание перевода (см. рис. 5.9). Очевидно, что начать запись в выходную ленту процессор сможет только тогда, когда ему будет доступен параметр цикла id. Когда процессор выполняет свёртку, используя правило грамматики с номером (3), то id входит в основу и Доступен процессору. Следовательно, в это правило мы можем включить операционный символ, который записывает на выходную ленту начальный участок перевода, вплоть до символа cns. Правило транслирующей грамматики с номером (3) будет выглядеть следующим образом:

(3)D → id = cns { OUT:= "id := cns; m1 : if id >" }.

При выполнении свёртки по этому правилу операционный символ записывает на выходную ленту параметр цикла id. Запись этой переменной в OUT может быть выполнена функцией "читать третий сверху символ магазина процессора и записать его на выходную ленту". Аналогичные действия ("Читать верхний символ магазина и записать его на выходную ленту") выполняются для начального значения параметра цикла cns. Остальные символы не зависят от входной цепочки (терминалы выходного языка) и просто помещаются на выходную ленту.

Рассуждая подобным образом, мы получим следующие правила транслирующей грамматики, содержащие элементы перевода (в порядке их использования при анализе входной цепочки и построении выходной Цепочки):

(4) E→ to cns{ OUT:= "cns then goto m2;" }

(7) B→ term{ OUT:= "term;" }

(5) F→ step cns{ OUT:= "id := id + cns; goto m1; m2 : " }

Добавив остальные правила грамматики, получим следующую транслирующую грамматику:

(1) S → A B C

(2) A → for D E F

(3) D → id = cns{ OUT:= "id := cns; m1 : if id >" }

(4) E → to cns{ OUT:= "cns then goto m2;" }

(5) F → step cns{ OUT:= "id := id + cns; goto m1; m2: " }

(6) B → S

(7) B → term{ OUT:= "term;" }

(8) C → next id

При восходящих методах синтаксического анализа (см. гл. 8, 9) строится обращённый правый вывод (разбор) цепочки - последовательность номеров правил, используемых при свёртках, - который для рассматриваемого примера равен: ((3), (4), (5), (2), (7), (8), (1)). При этом на выходной ленте формируется цепочка:

id := cns1; m1: if id > cns2 then goto m2; id := id + cns3; goto m1; m2: term;

Анализ полученной цепочки позволяет сделать следующие выводы:

1. Тело цикла (терминальный оператор term) включается неправильно (не перед увеличением параметра цикла на значение шага, а в конец выходной цепочки). Это связано с тем, что правило (5) применяется раньше, чем правило (7).

2. При использовании правила (5) значение параметра цикла id недоступно (id вытолкнут из магазина при свёртке по третьему правилу грамматики). Значение id должно быть передано в элемент перевода правила (5) для правильной генерации выходной цепочки.

3. При переводе вложенных циклов возникнет конфликт меток m1 и m2, т. к. m1 и m2 - фиксированные значения. Значения меток должны генерироваться при выполнении перевода и передаваться от места возникновения метки к месту её использования.

Ошибки, обнаруженные в выходной цепочке, можно легко устранить, расширив транслирующую грамматику до атрибутной транслирующей грамматики. Первую ошибку можно исправить несколькими способами:

- включить в правило (4) атрибут, в который записать координаты выходной цепочки, куда должно быть вставлено тело цикла, передать значение этого атрибута в правило (5) и использовать его для выполнения включения тела цикла в выходную Цепочку;
- связать с не терминалом F атрибут, значением которого является строка символов. При использовании правила (5) записать в эту строку операторы, реализующие изменение параметра цикла, затем передать значение этого атрибута в правило (1), где и переписать его значение в выходную цепочку;
- пополнить атрибутный процессор, выполняющий перевод, Дополнительной лентой, на которую записывать операторы, реализующие изменение переменной Цикла в правиле (5). В конце перевода при свёртке по правилу (1) сцепить вспомогательную ленту с выходной лентой.

Применение первых двух способов не вызывает принципиальных затруднений, но неэффективно при реализации процессора, поэтому используем последний способ устранения ошибки: на выходную ленту OUT1 будем за- писывать операторы установки начального значения параметра цикла, проверки завершения цикла и тело цикла, а на дополнительную ленту OUT2 - операторы изменения переменной цикла с заданным шагом.

Вторая и третья ошибки устраняются стандартным способом: включением в правила грамматики атрибутов и функций их вычисления.

Преобразуем транслирующую грамматику в транслирующую атрибутную грамматику.

Рассмотрим правило транслирующей грамматики:

D → id = cns{ OUT1:= "id := cns; m1 : if id >" }

При выполнении свёртки по этому правилу значения терминальных символов id и cns требуется передать из входной цепочки в выходную. Это реализуется передачей соответствующих атрибутов в операционный символ и ис- пользованием их при построении выходной цепочки:

$$D \rightarrow id_{a1} = cns_{a2} \ \{ \text{OUT1:= "}id_{n1} := cns_{n2} \text{ ; m1: if id >" } \}_{n1,\ n2}$$

$$s1, n1 \leftarrow a2; n2 \leftarrow a2$$

Запись "id l" в операционном символе означает, что в данное место выходной цепочки нужно поместить идентификатор, значение которого определяется унаследованным атрибутом n1 (значения id и cns - это не числовые значения этих объектов, а ссылки на таблицы, которые содержат необходимую информацию).

Так как переменная цикла id используется и в других правилах грамматики (правила (5) и (8)), то ее значение нужно передать в эти правила, связав с символом D атрибут и определив функцию для его вычисления. Окончательно правило грамматики примет вид:

(3) $D_{s1} \rightarrow id_{a1} = cns_{a2}$ { OUT1:= "id_{a1} := cns_{n2}; m1: if id >" }$_{n1,\ n2}$

s1, n1 ⟵ a1;
n2 ⟵a2

Для определения способа передачи значения переменной цикла id (атрибут a1) в правила (5) и (8) грамматики, рассмотрим фрагмент дерева грамматики, изображённый на рис. 5.5.

Процесс передачи значения лексемы id (атрибут a1) из правила (3) грамматики отмечен на рис. 5.5 сплошными линиями:

1. Из правой части правила (3) значение атрибута входного символа a1 присваивается синтезированному атрибуту s1 символа D из левой части этого же правила.
2. Из правой части правила (2) значение синтезированного атрибута s1 нетерминального символа D присваивается унаследованному атрибуту n1 не терминала F из правой части этого же правила.
3. Из левой части правила (5) значение унаследованного атрибута n1 не терминала F передаётся в правую часть унаследованному атрибуту n2 операционного символа.

Соответствующие описанному процессу правила АТ-Грамматики имеют вид:

(3) D_{s1}= id_{a1} { OUT1 := "id_{n1} := cns_{n2} ; m1: if id >" }$_{n1,\ n2}$
s1, n1 ⟵ a1;n2 ⟵ a2

(2) A → for D_{s1} E F_{n1}
n1 ⟵ s1

(5) F_{n1}→ step cns { OUT2 := " id := id +cns; goto m1; m2 : "}$_{n2}$
n2 ⟵ n1

Передача значения параметра цикла в правило (8) грамматики изображено на рис. 5.5 штриховыми линиями и описывается следующими правилами АТ-грамматики:

(2)
A_{s2} → for D_{s1} E F_{n1}
s2, n1 ⟵ s1

(1)
S → A_{s2} B C_{n1}
n1 ⟵ s2

(8) C_{n1} → next id { if n2 □ n3 then Error }
n2 ⟵ n1

Особенностью правила (8) построенной АТ-Грамматики является то, что его операционный символ должен интерпретироваться не как элемент цепочечного перевода, а как выполнение действий, напрямую не связанных с генерацией выходной цепочки. В этот операционный символ передаются значения параметра цикла из заголовка цикла и оператора его завершения. Если эти значения не равны, то должно возникать состояние ошибки процессора (Error).

Процесс генерации и передачи меток иллюстрируется на рис. 5.12. Метки m1 и m2 генерируются в операционных символах правил (3) и (4) грамматики соответственно с помощью процедуры-функции NewLabel, возвращающей при обращении к ней уникальное значение метки. Правила АТ-грамматики, описывающие процесс генерации меток, имеют вид:

(3) D_{s1}, s2 → id_{a1} = cns_{a2}

{ OUT1:= "id_{n1} := cns_{n2}; Label$_{s3}$: if id >" }$_{n1,\ n2,\ n3}$

s1, n1 ← 01;

n2 ← a2;s3 ← NewLabel;n3 ← s3;2 ← n3;

(4) E_{s3} → to cns { OUT1 := "cns_{then} goto $Label_{\text{sl}}$;" }n2

s1 ← NewLabel ;n2 ← s1;s3 ← n2;

(2) A → for $D_{s1,s2}$ E_{s3} $F_{n1,\ n2,\ n3}$

n1 ← s1;n3 ← s3

(5) $F_{n1,\ n2,\ n3}$ → step cns { OUT2 := "id := id + cns; goto Label$_{n4}$; Label$_{n5}$:"}$_{n4,\ n5}$

n4 ← n2;n5 ← n3

Сцепление выходных лент OUT1 и OUT2 должно быть выполнено один раз сразу после анализа и перевода всей входной цепочки. для реализации сцепления выходных лент грамматику необходимо пополнить новым (нулевым) правилом, включив в него операционный символ, выполнявши действия по конкатенации выходных лент. Результирующая АТ-грамматика будет иметь вид:

(0)S0 → S { OUT1 := OUT1 || OUT2 }

(1)S → A_{s1}B C_{n1}

n1 ← S1

(2)A_{s4} → for $D_{s1,s2}$ E_{s3} $F_{n1,\ n2,\ n3}$

s4, n1 ← s1;n2 ← s2;n3 ← s3;

(3) $D_{s1,s2}$ → id_{a1} = cns_{a2}

{ OUT1 := " id_{n1} := cns_{n2}; Label$_{s3}$: if id >" }$_{n1,\ n2,\ n3}$

s1, n1 ← 01; n2 ← 02; s3 ← NewLabel;n3 ← s3;s2 ← n3;

(4)E_{s3} → to cns_{a2}; { OUT1 := "cns_{n1} then goto Label$_{s1}$;" }$_{n1,\ n2}$

n1 ← 01;s1 ← NewLabel;

n2 ← s1;s3 ← n2;

(5)$F_{n1,\ n2,\ n3}$ → step cns_{a1}

{ OUT2 := "id_{n6} := id_{n6} + cns_{n7}; goto Label$_{n5}$; Label$_{n4}$:" }$_{n4,\ n5,\ n6}$

n4 ← n2;n5 ← n3;n6 ← n1;n7 ← a1;

(6)B → S

(7)B → term; { OUT1:= "term"; }

(8)C_{n1} → next id_{a1} { if n2 ≠ n3 then Error }$_{n4,\ n3}$

n2 ← n1;n3 ← a1

При переводе вложенных операторов цикла использовании двух выходных лент в процессоре и их сцепление при выполнении перевода может привести к ошибкам в структуре выходной цепочки. рассмотрим перевод следующей входной цепочки:

for i1 = c5 to c12 step c13

for i2 = c21 to c22 step c23

top

next i2

next i1

разбор которой в виде последовательности номеров правил и синтаксического дерева приведён на рис. 5.13.

К моменту использования правила (O) на выходных лентах OUT1 и OUT2 будут находиться следующие цепочки (на данном этапе проверки считаем, что вычисление атрибутов выполняется правильно):

OUT1:
i1 := c5;
Label5: if i1 > c12 then goto Label12;
i2 := c21;
Label21: if i1 > c22 then goto Label22; top;
OUT2:
i1 := i1 + c13; goto Label5; Label12:
i2 := i2 + c23; goto Label21; Label22:

После выполнения сцепления выходная (основная) лента будет содержать перевод:

OUT1:
i1 := c5;
Label5: if i1 > c12 then goto Label12;
i2 := c21;
Label21: if i1 > c22 then goto Label22; top;
OUT2:
i1 := i1 + c13; goto Label5; Label12:
i2 := i2 + c23; goto Label21; Label22:
i2 := c21;
Label21: if i1 > c22 then goto Label22;
top;
i1 := i1 + c13; goto Label5;
Label12:
i2 := i2 + c23;
goto Label21;
Label22:

Анализ выходной цепочки показывает, что перевод построен неверно: операторы завершения, внешнего и вложенного циклов следуют в обратном порядке. Связано это с тем, что вначале на выходную ленту OUT2 записываются операторы завершения внешнего цикла, и только после этого вложенного. При формировании же перевода на выходную ленту вначале должны быть записаны операторы завершения вложенного цикла, а затем - внешнего. Для того чтобы исправить эту ошибку, необходимо запись на ленту OUT2 и чтение из неё выполнять с одного конца, т. е. реализовать ею как стек с операциями PushOUT2 ("Втолкнуть на ленту OUT2") и PopOUT2 ("Вытолкнуть с ленты OUT2").

После включения операций PushOUT2 и PopOUT2 в правила (O) и (5) окончательно получим следующий вид атрибутной транслирующей грамматики:

(0) S0 → S { while (OUT2 not empty) OUT1 := OUT1 5 PopOUT2 }

(1)S → A_{s1} B C_{n1}

n1 ← s1

(2)

A_{s4} → for $D_{s1,s2}$ E_{s3} $F_{n1,\ n2,\ n3}$

s4, n1 ← s1;n2 ← s2;n3 ← s3;

(3)$D_{s1,s2}$ → id_{n1} = cns_{a1}

{ OUT1 := "id_{n1} := cns_{n1}; Label$_{s3}$: if id >" }$_{n1,\ n2,\ n3}$

s1, n1 ← a1;n2 ← a2;s3 ← NewLabel;n3 ← s3;s2 ← n3;

(4)E_{s3} → to cns_{a1} { OUT1 := "cnsnl then goto Label$_{s3}$;" }$_{n1,\ n2}$

n1 ← a1;s1 ← NewLabel;n2 ← s1;s3 ← n2;

(5) $F_{n1,\ n2,\ n3}$ → step cns_{a1}

{ PushOUT2("id := "id_{n6} := id_{n6} + cns_{n7}; goto Label$_{n5}$; Label$_{n4}$:" :") }$_{n4,\ n5,\ n6}$

(6) B → 8

(7)B → term; { OUT1 := "term; }

(8)C_{n1} → next id_{a1} { if n2 ≠ n3 then Error }$_{n2,\ n3}$

n2 ← n1;n3 ← a1

Читателям рекомендуется самостоятельно проверить правильность окончательного варианта построения АТ-Грамматики для различных тестовых входных цепочек.

Контрольные вопросы

1.Как определяется понятие семантики языков программирования?

2.Как можно описать простейшие переводы?

3.Какие переводы называются синтаксически управляемыми?

4.Что такое элемент перевода?

5.Дайте определение СУ-схемы. Какие цепочки называются входными и выходными?

6.Как определить входную и выходную грамматики СУ-схемы?

7.Как происходит порождение пар цепочек под управлением СУ-схемы?

8.Как происходит преобразование деревьев под управлением СУ-схем?

9.Какие СУ-схемы называются простыми?

1О. Какие переводы позволяют описать простые СУ-схемы?

11.Какая грамматика называется транслирующей гамматикой?

12.Что такое входная и выходная грамматики транслирующей грамматики?

13. Как определяется активная цепочка, ее входная и операционная части?

14. Как связаны вывод цепочки и дерево вывода активной цепочки?

15. Какая Грамматика называется атрибутной транслирующей грамматикой?

16. Какие атрибуты называются унаследованными и синтезированными?

17. В чем основные различия между унаследованными и синтезированными атрибутами?

18. Что представляет собой правило вычисления атрибутов?

19. Как строится граф зависимостей?

2О. Как граф зависимостей используется при вычислении атрибутов?

21.Какие виды объектов включаются в выходную цепочку при переводе?

22.Из каких этапов состоит процесс построения транслирующей грамматики?

23.Из каких этапов состоит процесс построения АТ-Грамматики?

24. Какие типы ошибок позволяет найти тестирование АТ-Грамматики?

Упражнения

1.Разработайте простую СУ-схему, описывающую перевод арифметических скобочных выражений, содержащих операции '+' и '*' в:

1.1.постфиксную запись.

1.2.префиксную запись.

2.В языке ALGOL-6O выражения строятся с помощью операций, имеющих следующие приоритеты (в порядке убывания):

(1)↑(4)≤ < = ≠ > ≥(7) ∨

(2)*/ ÷(5) ¬(8) →

(3)+ -(6)∧(9) ≡

Разработайте простую СУ-схему, описывающую перевод инфиксных выражений, содержащих перечисленные операции, в постфиксную запись.

3.Постройте МП-преобразователи, реализующие СУ-схемы из упр. 1.

4.Для простой СУ-схемы:

E → + EE, EE +

E → * EE, EE *

E → a, a

постройте эквивалентный МП-преобразователь.

5.Для заданной СУ-схемы:

8 → a $S^{(1)}$ b A c $S^{(2)}$ d $S^{(3)}$, e A c $S^{(2)}$ d $S^{(3)}$ f $S^{(1)}$ g

S → i, i

A →i, i

постройте переводы для входных цепочек:

5.1.aibicd .

5.2.aaibicidibicid .

6.Постройте вывод в транслирующей грамматике G0T цепочек:

6.1.i* (i+i).

6.2.(i+i) * (i + i).

7.Определите транслирующую грамматику, допускающую в качестве входа произвольную цепочку из нулей и единиц и порождающую на выходе:

7.1.обращение входной цепочки.

7.2.цепочку 0n1m, где n - число нулей, а m - число единиц во входной цепочке.

8.Постройте транслирующую грамматику, определяющую перевод логических выражений, составленных из логических переменных, скобок и знаков операций дизъюнкции, конъюнкции и отрицания:

8.1.из инфиксной записи в ПОЛИЗ.

8.2.из ПОЛИЗ в инфиксную запись.

8.3.из инфиксной записи в функциональную запись такую, например, что выражение a ∩ (b ∪ c) будет представлено в виде f∩ (a, f∪ (b, c)), где f∩ и f∪ - отдельные символы.

9.Постройте АТ-Грамматику, описывающую перевод оператора присваивания некоторого Гипотетического языка программирования в цепочку тетради с кодами операций: ПРИСВОИТЬ, СЛОЖИТЬ, ВЫЧЕСТЬ, УМНОЖИТЬ, ДЕЛИТЬ. Левой частью оператора присваивания является идентификатор, а правой частью бесскобочное арифметическое выражение, выполняемое справа налево в порядке написания операций. В арифметическом выражении можно использовать идентификаторы и знаки арифметических операций: +, -, *, /.

10. Постройте АТ-грамматику, описывающую перевод оператора присваивания некоторого гипотетического языка программирования в цепочку тетрад с кодами операций: ПРИСВОИТЬ, И, ИЛИ. НЕ. левой частью оператора присваивания является идентификатор, а правая часть представляет собой логическое выражение, составленное из идентификаторов, скобок и знаков логических операций: ∩, ∪, ¬. порядок выполнения логического выражения определяется обычным приоритетом логических операций.

11. Постройте АТ-грамматику, описывающую перевод в тетрадку с кодом операции ПЕРЕХОД_ПО_РАВНО условного оператора, которому соответствует следующее правило входной Грамматики:

условный оператор → if = goto .

Лексема представляет собой номер оператора, к которому осуществляется переход в случае равенства, а - идентификатор.

12. Для входной грамматики, описывающей арифметические выражения, с правилами вывода:

S → EE → (E)

E → E := EE → i

E → E + E

где - лексема, значением которой является указатель на элемент таблицы идентификаторов, а семантика выражений такая же, как в языке С, постройте АТ-грамматику, которая проверяет, представляет ли левая Часть выражения значение, и, в случае успеха, стройте ПОЛИЗ выражения.

13. Для входной грамматики, описывающей объявление переменных, с правилами вывода:

(1)D→ L(4)→ int

(2)L→ , L(5)→ real

(3)L→ :

постройте АТ-грамматику, которая заносит значение типа каждого идентификатора в таблицу идентификаторов.

14. Для входной грамматики, описывающей арифметические выражения, с правилами вывода:

E → (E + E)

E → (E * E)

E → i

где - лексема, значением которой является указатель на элемент таблицы идентификаторов, определить постфиксную 8-атрибутную грамматику,

описывающую перевод этих выражений в цепочку тетради с концами операций: СЛОЖИТЬ и УМНОЖИТЬ, и построить 8-атрибутный ДМП-процессор, выполняющий заданный перевод.

Глава 6. Разработка и реализация синтаксически управляемого перевода

Рассмотрим два подкласса корректных АТ-грамматик, которые часто используются при проектировании языковых процессоров: L-атрибутные транслирующие грамматики и S-атрибутные транслирующие грамматики.

1. L-атрибутные и S-атрибутные транслирующие грамматики

Определение: АТ-грамматика называется L-атрибутной тогда и только тогда, когда выполняются следующие условия:

1. Аргументами правила вычисления значения унаследованного атрибута символа из правой части правила вывода могут быть только унаследованные атрибуты символа из левой части и произвольные атрибуты символов из правой части, расположенные левее рассматриваемого символа.
2. Аргументами правила вычисления значения синтезированного атрибута символа из левой части правила вывода являются унаследованные атрибуты этого символа или произвольные атрибуты символов из правой части.
3. Аргументами правила вычисления значения синтезированного атрибута операционного символа могут быть только унаследованные атрибуты этого символа.

Является ли произвольная АТ-грамматика L-атрибутной, можно проверить, независимо исследуя каждое правило вывода и каждое правило вычисления значения атрибута. Примером L-атрибутной транслирующей грамматики является грамматика G^A.

При выполнении условия 1 определения унаследованные атрибуты каждой вершины дерева вывода зависят (непосредственно или косвенно) только от атрибутов входных символов, расположенных в дереве левее данной вершины, что позволяет использовать L-атрибутные грамматики в нисходящих синтаксических анализаторах. Условия 2 и 3 введены с целью сделать АТ-грамматику корректной.

В L-атрибутной транслирующей грамматике атрибуты символов A, B и C из правила вывода

A →BC можно вычислять в следующем порядке:

унаследованные атрибуты символа A;
унаследованные атрибуты символа B;
синтезированные атрибуты символа B;
унаследованные атрибуты символа C;
синтезированные атрибуты символа C;
синтезированные атрибуты символа A.

Определение: АТ-грамматика называется S-атрибутной тогда и только тогда, когда она является L-атрибутной и все атрибуты нетерминалов синтезированные.

Ограничения, накладываемые на L-атрибутную транслирующую грамматику, позволяют вычислять значения атрибутов в процессе нисходящего анализа входной цепочки. Нисходящий етерминированный анализатор для LL(1)-грамматик требует, чтобы L-атрибутная транслирующая грамматика, описывающая перевод, имела форму простого присваивания.

6.2 Форма простого присваивания

При определении АТ-грамматики в правила вычисления атрибутов записывались в виде операторов присваивания, левые части которых представляют собой атрибут или список атрибутов, а правые части — функцию, использующую в качестве аргументов значения некоторых атрибутов. Простейший случай функции из правой части оператора присваивания — тождественная или константная функция, например a ← b или x, y←1.

Определение: правило вычисления атрибутов называется копирующим правилом тогда и только тогда, когда левая часть правила — это атрибут или список атрибутов, а правая часть – константа или атрибут. Правая часть называется источником копирующего правила, а каждый атрибут из левой части – приемником копирующего правила.

Если источники нескольких копирующих правил совпадают, то их приемники можно объединить в одну левую часть. Например, правила z, w← a и x, y←z можно записать в виде: x, y, z, w← a , т. к. источнику второго правила z, согласно первому правилу, присваивается значение a. Аналогично x←y и a, b ← y можно записать как a, b, x ←y.

Определение: множество копирующих правил называется независимым, если источник каждого правила из этого множества не входит ни в одно из других правил множества.

Если два копирующих правила независимы, их нельзя объединять.

Определение: атрибутная транслирующая грамматика имеет форму простого присваивания тогда и только тогда, когда:

атрибутов операционных символов.

копирующих правил независимо.

Примером АТ-грамматики в форме простого присваивания является грамматика G4.

Рассмотрим процедуру преобразования произвольной L-атрибутной транслирующей грамматики в эквивалентную L-атрибутную грамматику в форме простого присваивания. Смысл используемого здесь понятия эквивалентности объясняется далее:

Каждой функции $f(x_1, x_2, \ldots, x_n)$, входящей в правило вычисления атрибутов, связанное с некоторым правилом вывода грамматики, создать соответствующий ей операционный символ $\{f\}$, который определяется следующим образом:

$\{f\}$ x_1 x_2, ... , x_n, p унаследованные $x_1, x_2, \ldots, x_n$

синтезированный p

p← $f(x_1, x_2, \ldots, x_n)$

Для каждого некопирующего правила z_i, $z_2, \ldots, z_b \leftarrow f(y_1, y_2, \ldots, y_n)$ связанного с некоторым правилом вывода грамматики, найти символы a1 ... , a„, которые не содержатся в этом правиле вывода, и вставить в его правую часть символ$\{f\}a_n \leftarrow a_2, \ldots, a_n$. Заменить некопирующее правило на следующие (n + 1) копирующих правил:

$a_i \leftarrow y_n$ для каждого аргумента y_i,

$z_1, z_2, \ldots, z_m \leftarrow r$

При включении в правило вывода операционного символа необходимо соблюдать следующие условия:

операционный символ должен располагаться правее всех символов правой части правила вывода, атрибутами которых являются аргументы $y_1, y_2, \ldots, y_n$;

операционный символ должен располагаться левее всех символов правой части правила вывода, атрибутами которых служат $z_1, z_2, \ldots, z_m$;

С учетом предыдущих ограничений опреационный символ может быть вставлен в любое место правой части правила вывода, но предпочтение следует отдать самой левой из возможных позиций, т. к. это позволяет упростить реализацию синтаксического анализатора.

Два копирующих правила, соответствующие одному и тому же правилу вывода, необходимо объединить, если источник одного из них входит в другое. Это достигается удалением правила с лишним источником и объединением его приемников с приемниками оставшегося правила.

Если в качестве источника копирующего правила используется константная функция, являющаяся процедурой-функцией без параметров (например, функция GETNEW из АТ-грамматики), то такие атрибутные правила объединять нельзя, т. к. два разных вызова функции без параметров могут давать разные значения.

Рассмотрим пример преобразования L-атрибутной транслирующей грамматики, порождающей префиксные арифметические выражения над константами, в форму простого присваивания. Атрибутные правила вывода этой грамматики имеют следующий вид:

E_p синтезированный p
{ОТВЕТ}p, унаследованный r
$S \rightarrow E_p$ {ОТВЕТ}r
$r \leftarrow p$
$E_p \rightarrow + E_q E_p$
$p \leftarrow q + r$
$E_p \rightarrow *E_q E_r$
$p \leftarrow q* r$
$E_p \rightarrow c_r$
$p \leftarrow r$

Входными символами грамматики являются: лексема c, представляющая собой целочисленную константу, и знаки арифметических операций: «+» и «*». Входной символ c имеет один атрибут, значением которого является значение константы. Нетерминальный символ E и операционный символ {ОТВЕТ} также имеют по одному атрибуту. Значением синтезированного атрибута символа E является значение подвыражения, порождаемого этим символом, а значением унаследованного атрибута символа {ОТВЕТ} — значение всего выражения, порождаемого грамматикой.

Исходная грамматика содержит два некопирующих правила: $p \leftarrow q + r$ и $p \leftarrow q* r$, правые части которых представляют собой функции сложения и умножения соответственно. Для преобразования заданной грамматики в форму простого присваивания введем операционные символы {СЛОЖИТЬ}A, B, R и {УМНОЖИТЬЦ}A, B, R каждый из которых имеет по два унаследованных ат-

рибута A и B и один синтезированный атрибут R. Для операционного символа {СЛОЖИТЬ}A, B, R атрибутное правило имеет вид: R ← A + B, а для операционного символа {УМНОЖИТЬ}A, B, R — R ←A X B.

Для того чтобы преобразованная грамматика также была L-атрибутной, символ {СЛОЖИТЬ} необходимо поместить правее всех символов правой части правила вывода (2), т. к. одним из аргументов сложения является атрибут самого правого символа E. Атрибут, получающий в качестве своего значения результат сложения, в определении места расположения символа {СЛОЖИТЬ} не участвует, т. к. он не приписан ни к одному из символов правой части. Аналогичные рассуждения относительно операционного символа {УМНОЖИТЬ} определяют крайнюю правую позицию правой части правила (3), как единственно возможное место расположения этого символа. Полученная в результате преобразования L-атрибутная грамматика в форме простого присваивания имеет вид:

Ep синтезированный p
{ОТВЕТ}r, унаследованный r
{СЛОЖИТЬ}A, B, R унаследованный A, B
R ← A + B синтезированный R
{УМНОЖИТЬ}A, B, R унаследованный A, B
R ←A* B синтезированный R
$S \rightarrow E_p$ {ОТВЕТ}r
r ← p
$E_p \rightarrow + E_q E_p$ {СЛОЖИТЬ}A, B, R
A← q
B ←r
P ←R
$E_p \rightarrow * E_q E_p$ {УМНОЖИТЬ}A, B, R
A←q
B← r
p ←R
$E_p \rightarrow c_r$
p←r

Эта грамматика порождает те же входные цепочки и значение синтезированного атрибута нетерминала E, что и исходная грамматика. Однако формально она не определяет того же самого перевода, т. к. преобразованные правила (2) и (3) удлиняют активную цепочку, включив в нее действия, обеспечивающие вычисление функций сложения и умножения соответственно. Для того чтобы преобразованная грамматика определяла тот же перевод, что и исходная, введенные в процессе преобразования операционные символы не следует выдавать в выходную цепочку.

3. Атрибутный перевод для LL(1)-грамматик

Расширим "1-предсказывающий" алгоритм разбора так, чтобы он мог выполнять атрибутный перевод, определяемый L-атрибутной транслирующей грамматикой, входной грамматикой которой является LL(1)-

грамматика. Моделирование такого алгоритма можно осуществить с помощью атрибутного **ДМП-преобразователя** с концевым маркером.

Сначала рассмотрим проблему выполнения синтаксически управляемого перевода, определяемого транслирующей грамматикой цепочечного перевода, входной грамматикой которого является LL(1)-грамматика.

3.1 Реализация синтаксически управляемого перевода для транслирующей грамматики

Преобразуем "1-предсказывающий" алгоритм разбора для LL(1)-грамматик, включив в него действия, обеспечивающие перевод входной цепочки, порождаемой входной грамматикой, в цепочку операционных символов и запись этой цепочки на выходную ленту. Преобразованный таким образом алгоритм в дальнейшем будем называть нисходящим детерминированным процессором с магазинной памятью (нисходящий ДМП-процессор). При этом, если из контекста ясно, что речь идет о нисходящем методе анализа, слово "нисходящий" будем опускать.

В транслирующей грамматике множество терминальных символов разбито на множество входных символов $\sum_i$, и множество операционных символов $\sum_a$.

Расширим алфавит магазинных символов, добавив в него операционные символы. Тогда $V_p = \sum_i \cup \sum_a \cup N$. Затем доопределим управляющую таблицу M, которая для транслирующей грамматики цепочечного перевода задает отображение множества $(V_p \cup \{\perp\})$ X $(\sum I \cup \{\in\})$ в множество, состоящее из следующих элементов:

(β, у), где $\beta \in V_p^*$ — цепочка из правой части правила транслирующей грамматики A →yβ, a у $\in \sum_a^*$;

ВЫДАЧА (X), где X $\in \sum_a$

ВЫБРОС;

ДОПУСК;

ОШИБКА.

Пусть FIRST(x) = a, где x — неиспользованная часть входной цепочки. Тогда работу ДМП-процессора в зависимости от элемента управляющей таблицы M(X, o) можно определить следующим образом:

1. (x, Xα, π) ⊢ (x, βα), если M(X, a) = (β, y). Верхний символ магазина X заменяется цепочкой β ∈ Vp*, и в выходную цепочку дописывается цепочка операционных символов y. Входная головка при этом не сдвигается.

2. (x, Xα, π) ⊢(x, α, πX), если M(X, a) = ВЫДАЧА{X). Это означает, что если верхний символ магазина — операционный символ, то он выталкивается из магазина и записывается на выходную ленту. Входная головка не сдвигается.

Действия, соответствующие элементам управляющей таблицы: ВЫБРОС, ДОПУСК и ОШИБКА, остаются теми же, что и в "1-предсказывающем" алгоритме для LL(1)-грамматик.

Опишем алгоритм построения управляющей таблицы M.

Алгоритм построения управляющей таблицы для транслирующей грамматики

цепочечного перевода, входной грамматикой которой является LL(1)-грамматика

Вход: Транслирующая грамматика цепочечного перевода $G^T = (N, \sum_i, \sum_a, P, S)$, входная грамматика которой является LL(1)- грамматика.

Выход : Корректная управляющая таблица M для грамматики GT.

Описание алгоритма:

Управляющая таблица M определяется на множестве $(N \cup \sum_i \cup \sum_a \cup \{\perp\}) \times \sum_i \cup \{\varepsilon\})$ по следующим правилам:

Если $A \rightarrow y\beta$ - правило вывода грамматики GT, где $y \in \sum_a^*$, а y — либо ε, либо цепочка, начинающаяся с терминала или нетерминала, то M{A, a) = (β, y) для всех $a \neq \varepsilon$, принадлежащих множеству FIRST(β).

Если $\varepsilon \in$ FIRST(β), то M(A, b) = (β, y) для всех $b \in$ FOLLOW (A).

Заметим, что при вычислении FIRST(β) операционные символы, входящие в цепочку β, вычеркиваются.

$M(X, a)$ = ВЫДАЧА(X) для всех $x \in \sum_a$ И $a \in \sum_i \cup \{\varepsilon\}$.

M(a, a) = ВЫБРОС для всех $a \in \sum_i$.

M (⊥ , ε) = допуск

В остальных случаях M(X, a) = ОШИБКА для $X \in (N \cup \sum_i \cup \sum_a \cup \{\perp\})$ и $a \in \sum_i \cup \{ \varepsilon \}$.

Построим управляющую таблицу для транслирующей грамматики, описывающей перевод инфиксных арифметических выражений в ПОЛИЗ. Эта транслирующая грамматика получена из входной LL(1)-грамматики G1 путем включения в нее операционных символов {i}, {+} и {*}:

E →E ′(5)T′ →* P{*} T′

E ′ →+ T{+} E′6)T′ →ε

E ′ →ε

(7)P →i {i}

T →PT′

(8)P →(E)

Управляющая таблица должна содержать 14 строк, помеченных символами из множества $N \cup \sum i \cup \sum a \cup \{\perp\}$, и 6 столбцов, помеченных символами из множества $\sum_i \cup \{ \varepsilon \}$.

Построение управляющей таблицы для строк, отмеченных символами из множества $N \cup \sum i \cup \{\perp\}$, ничем не отличается от построения таблицы для соответствующей входной LL(1)-грамматики (табл. 1.1), а строки управляющей таблицы, отмеченные операционными символами, содержат значения ВЫДАЧА{{X}), где X ∈{{i}, {+},{*}}. Заметим, что элементы таблицы, соответствующие строкам, помеченным операционными символами, для всех столбцов одинаковые, т. к. действия, выполняемые ДМП-процессором в случае, когда верхним символом магазина является операционный символ, не зависят от входного символа.

Таблица 1.1. Управляющая таблица M для транслирующей грамматики, описывающей перевод инфиксных арифметических выражений в ПОЛИЗ

	i	(	)	+	*	ε
E	TE’,ε	TE’,ε				

E’			ε, ε	*P{*} T’, ε		ε, ε
T	PE’,ε	PE’,ε				
T’			ε, ε	ε, ε	*P{*} T’, ε	ε, ε
P	i{i},ε	(E),ε				
i	ВЫБРОС					
(		ВЫБРОС				
)			ВЫБРОС			
+				ВЫБРОС		
*					ВЫБРОС	
⊥						ДОПУСК
{i}	ВЫДАЧА({i})					
{+}	ВЫДАЧА({+})					
{*}	ВЫДАЧА({*})					
Начальное содержимое магазина - E⊥						

Начальное содержимое магазина — E⊥

Для входной цепочки i +i * i ДМП-процессор проделает следующую последовательность тактов:

(i +i * i , E⊥, ε) ⊢ (i +i * i , TE'⊥, ε)
⊢ (i +i * i , PT'E'⊥, ε)
⊢(i +i * i , {I} T'E'⊥, ε
⊢ (+ i * i , {i}T'E'⊥, ε)
⊢ (+ i * i , {i}T'E'⊥, {i})
⊢ (+i * i , {i}E'⊥, {i})
⊢ (+i * i , +T {+} E'⊥,{i})
⊢ (i * i , T {+} E'⊥,{i})
⊢ (i * i , PT '{+} E'⊥, {i})
⊢ (i * i , i {i} T '{+} E'⊥, {i})
⊢ (*i , {i} T '{+} E'⊥, {i})
⊢ (* i , T '{+} E'⊥, {i} {i})
⊢ (* i , *P{*}T'{+}E'⊥, {i} {i})
⊢ (i , P{*}T'{+}E'⊥, {i} {i})
⊢ (i , i {i} {*}T'{+}E'⊥, {i} {i})
⊢ (ε , {i}{*}T'{+}E'⊥, {i} {i})
⊢ (ε , {i}{*}T'{+}E'⊥, {i} {i})
⊢ (ε , T'{+}E'⊥, {i} {i}{i}{*})
⊢ (ε , {+}E'⊥, {i} {i}{i}{*})
⊢ (ε , E'⊥, {i} {i}{i}{*}{+})
⊢ (ε , ⊥, {i} {i}{i}{*}{+})

ДМП-процессор можно использовать в качестве базового процессора для других видов синтаксически управляемого перевода, если операцию выдачи

операционного символа в выходную ленту заменить операциями вызова соответствующих семантических процедур.

3.2 L-атрибутный ДМП-процессор

Процедура преобразования ДМП-процессора в атрибутный ДМП-процессор, которая описывается в данном пособии, требует, чтобы **АТ-грамматика, определяющая перевод, имела форму простого присваивания.**

Рассмотрим построение L-атрибутного ДМП-процессора, реализующего перевод, определяемый L-атрибутной транслирующей грамматикой в форме простого присваивания, входной грамматикой которого является LL(1)-грамматика.

Сначала построим ДМП-процессор, реализующий цепочечный перевод, описываемый транслирующей грамматикой цепочечного перевода, которая получается из заданной L-атрибутной транслирующей грамматики после удаления из нее всех атрибутов. Затем расширим полученный таким образом ДМП-процессор, включив для каждого магазинного символа множество полей для представления атрибутов символа и дополнив управляющую таблицу действиями по вычислению атрибутов и записи их в соответствующие поля.

Для удобства изложения будем считать, что магазинный символ с п атрибутами представляется в магазине (n + 1)-ой ячейками, верхняя из которых содержит имя символа, а остальные — поля для атрибутов. Поля магазинного символа доступны для записи и извлечения атрибутов от момента вталкивания символа в магазин до момента выталкивания его из магазина.

В момент вталкивания символа в магазин в поле для каждого синтезированного атрибута и атрибута входного символа заносится указатель на связанный список полей, соответствующих унаследованным атрибутам, где этот атрибут должен запоминаться, а поле для каждого унаследованного атрибута остается пустым. Содержимое полей синтезированных атрибутов и атрибутов входных символов остается неизменным в течение всего времени нахождения символа в магазине, а поля, соответствующие унаследованным атрибутам, приобретают значения атрибутов к моменту времени, когда магазинный символ окажется в верхушке магазина.

Пусть x— остаток входной цепочки, и FIRST(x) = a. Опишем действия, которые должен выполнять L-атрибутный ДМП-процессор в зависимости от элемента управляющей таблицы M(X, o), где X— символ в верхушке магазина.

Начальная конфигурация. В магазине находится маркер дна и начальный символ грамматики. Поля начального символа грамматики, соответствующие унаследованным атрибутам, заполняются начальными значениями атрибутов, которые задаются L-атрибутной транслирующей грамматикой, а в поля, соответствующие синтезированным атрибутам, заносятся пустые указатели, которые служат маркерами конца списков.

M(X, a) = ВЫБРОС (символ в верхушке магазина совпадает с текущим входным символом). В этом случае каждое поле верхнего магазинного символа содержит указатель на список полей магазина, в которых требуется поместить значение атрибута текущего входного символа. Операция ВЫБРОС расширяется таким образом, что каждый атрибут текущего входного символа

копируется во все поля списка на который указывает соответствующее поле верхнего магазинного символа.

M(X, a) = ВЫДАЧА(X) (в верхушке магазина находится операционный символ). Операция ВЫДАЧА(X) ДМП-процессора расширяется следующим образом:

унаследованных атрибутов извлекаются из соответствующих полей верхнего магазинного символа и используются затем при выдаче символа в выходную цепочку. При этом следует помнить, что если операционный символ появился в результате преобразования исходной атрибутной транслирующей грамматики в форму простого присваивания, то такой символ в выходную цепочку не выдается;

значение синтезированных атрибутов вычисляются по правилам вычисления атрибутов, связанным с данным операционным символом, после чего значение каждого синтезированного атрибута помещается во все поля списка, на который указывает соответствующее поле символа из верхушки магазина.

M(X, a) = (β, y) (в верхушке магазина находится нетерминал). В этом случае L-атрибутный ДМП-процессор вталкивает в магазин цепочку символов β из правой части распознаваемого правила B DSLFTN WTGJXRE операционных символов y. При этом вычисляются атрибуты операционных символов, которые не вталкиваются в магазин, и заполняются поля атрибутов магазинных символов и символов цепочки β, вталкиваемых в магазин.

Источниками атрибутных правил, связанных с правилами вывода L-атрибутной транслирующей грамматики в форме простого присваивания, могут быть только константы, унаследованные атрибуты нетерминалов из левой части правил вывода, атрибуты входных и операционных символов, синтезированные атрибуты нетерминалов из правой части правил вывода. В табл. 1.2 приведены значения источников копирующих правил в момент времени, когда верхним символом магазина является нетерминалы унаследованные атрибуты символов из правой части правил вывода и унаследованные атрибуты символов из правой части правил вывода. В табл. 1.3 приведены поля магазина, соответствующие приемникам атрибутных правил, которые необходимо заполнить во время перехода L-атрибутного ДМП-процессора при M{X, a) = (β, y).

При выполнении перехода L-атрибутный ДМП-процессор выполняет следующие атрибутные действия:

Вычисляет значения синтезированных атрибутов операционных символов, которые не вталкиваются в магазин, и выдает цепочку операционных символов с их атрибутами в выходную цепочку, если эти символы не появились в результате преобразования грамматики в форму простого присваивания.

Если источник копирующего правила доступен, то значение источника помещается в соответствующее поле магазинного символа.

Если источник копирующего правила недоступен, то после вталкивания символа в магазин в соответствующие поля символа заносятся указатели на список полей, где будут храниться значения унаследованных атрибутов.

Действия L-атрибутного ДМП-процессора для элементов управляющей таблицы, имеющих значения ДОПУСК и ОШИБКА, остаются теми же самыми, что у ДМП-процессора.

Построим L-атрибутный ДМП-процессор для L-атрибутной транслирующей грамматики в форме простого присваивания.

На рис. 1.4 показано представление полей магазинных символов, а в табл. 1.5 приведена управляющая таблица для транслирующей грамматики, полученной из исходной L-атрибутной транслирующей грамматики путем вычеркивания из нее атрибутов.

Таблица 1.5. Управляющая таблица для транслирующей грамматики, полученной из исходной грамматики

S	i:=E{:=},				
E	iR, ε				
R			+ i{+}	*i{*} R,	ε, ε
/	ВЫБРОС				
:=		ВЫБР			
+			ВЫБРО		
*				ВЫБРО	
⊥					ДОПУСК
{:=	ВЫДАЧА({:=}p q)				
{+}	ВЫДАЧА({+}P, q. г)				
{*}	ВЫДАЧА({*}P. q. г)				

Начальное содержимое магазина — S⊥

4. Атрибутный перевод методом рекурсивного спуска

Сначала рассмотрим, каким образом можно изменить процедуры для распознавания цепочек, порождаемых нетерминальными символами грамматики, для реализации перевода, описываемого транслирующей грамматикой цепочечного перевода.

В этом случае правила составления процедур дополняются следующим правилом: если текущим символом правой части правила вывода грамматики является операционный символ Y, ему соответствует вызов процедуры записи операционного символа в выходную цепочку output (Y).

В качестве примера реализации перевода с использованием метода рекурсивного спуска рассмотрим транслирующую грамматику, описывающую перевод инфиксных арифметических выражений в польскую инверсную запись. Эта грамматика построена на основе входной грамматики G1 и имеет следующие правила:

$S \rightarrow E\perp$

$E \rightarrow TE'$

$E' \rightarrow +T\{+\}\ E' \mid \varepsilon$

$T \rightarrow PT'$

$T' \rightarrow *P\{*\}\ T' \mid \varepsilon$

$P \rightarrow (E) \mid i\ \{i\}$

Головной модуль Recurs_Method и процедуры для распознавания нетерминальных символов E (Proc_E) и T (Proc_T) не изменятся. Процедура на языке

Pascal, реализующая перевод для транслирующих грамматик методом рекурсивного спуска приведена в листинге 1.1.

Листинг 1.1

```
Procedure  Recurs_Method_TG (List_Token: tList, List_Oper: tListOper);
        {List_Token —входная цепочка,
         List_Oper — выходная цепочка}
Procedure  Proc_E
begin
  Proc_T;
Proc_E1
  end {Proc_E};
Procedure Proc_E1;
   begin
if Symb = '+' then
begin

NextSymb;

Proc_T;
Output ({+});
Proc_E1
end
end {Proc_E1}
Procedure Proc_T;
begin
Proc_P;
Proc_T1
end {Proc_T};
Procedure Proc_T1;
begin
if  Symb = '+' then begin
NextSymb;
Proc_P;
Output({*});
  Proc_T1
end
end {Proc_T1};
Procedure Proc_P;
 begin
if  Symb = '(' then

begin
    NextSymb;
     Proc_E;
if  Symb = ')' then
                  NextSymb;
```

```
else
   Error
                end
                else
   if  Symb = 'i' then

begin
     NextSymb;
     Output({i})
   end

else
   Error
   end;  {Proc_P}

begin
      {В начале анализа переменная Symb содержит первый символ цепочки}
   Proc_E;
   if  Symb = '⊥' then Access
   else
     Error
   end {Recurs_Method_TG};
```

На рис. 1.2 приведен список имен процедур в порядке их вызова при переводе входной цепочки (i + i) * i ⊥. в предпоследнем столбце в строке, соответствующей вызову процедуры NextSymb, изображена непрочитанная часть входной цепочки (текущий символ находится слева) непосредственно после вызова этой процедуры, а в последнем столбце в строке, соответствующей вызову процедуры Output(Y), изображена часть выходной цепочки, построенной непосредственно после вызова процедуры Output(Y).

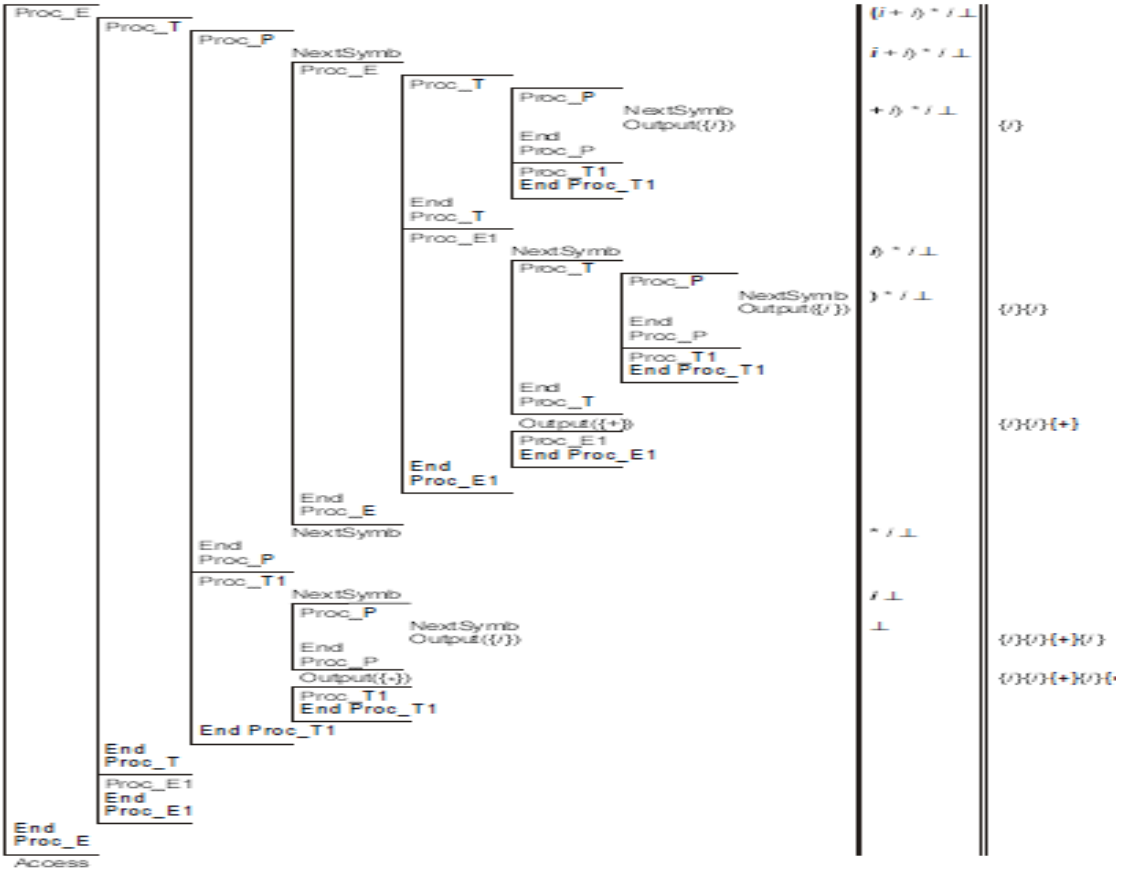


Рисунок 1.2. Порядок вызова процедур при переводе цепочки (i + i) * i ⊥Для того, чтобы методом рекурсивного спуска можно было выполнять L-атрибутный перевод, введем в процедуры распознавания нетерминальных символов параметр для каждого атрибута этого символа. При этом если параметру процедуры соответствует унаследованный атрибут нетерминального символа, то вызов этой процедуры производится с фактическим параметром — значением унаследованного атрибута, а в случае синтезированного атрибута в качестве фактического параметра используется переменная, которой должно быть присвоено значение синтезированного атрибута в момент выхода из процедуры.

Для того чтобы процедуры обеспечивали правильную передачу параметров для унаследованных и синтезированных атрибутов, они должны быть написаны на языке программирования, который обеспечивает способы передачи параметров "вызов по значению" (для унаследованных атрибутов) и "вызов по ссылке" (для синтезированных атрибутов).

Замечание. Язык программирования Pascal поддерживает оба метода передачи параметров, а в языке C имеется единственный механизм передачи параметров — "вызов по значению" [36]. Эффект вызова по ссылке в языке C может быть получен, если использовать в качестве параметров указатели.

Поскольку имена атрибутов нетерминальных символов используются в качестве параметров процедур для распознавания этих нетерминалов, при именовании атрибутов необходимо выполнять дополнительное требование, заключающееся в том, что все вхождения некоторого нетерминала в левые части правил вывода АТ-грамматики должны иметь один и тот же список имен атрибутов. Например, в грамматике не может быть таких правил вывода:

$R_{p_1,\ t_2} \rightarrow +i_{q_1}\ \{*\}_{p_2,\ q_2,\ r_1}\ R_{r_1,\ t_1}$
$R_{p_1,\ t_2} \rightarrow +i_{q_1}\ \{*\}_{p_2,\ q_2,\ r_1}\ R_{r_1,\ t_1}$
$R_{p_1,\ p_2} \rightarrow \varepsilon$

т. к. первые два вхождения нетерминала R имеют атрибуты $p_1\ t_2$, а последнее вхождение — p_1, p_2 В этом случае необходимо выбрать какой-то один список имен атрибутов нетерминала R (например, p и t), использовать эти имена при описании типа атрибутов этого нетерминала (например, унаследованный p, синтезированный t) и переименовать его атрибуты соответствующим образом.

Указанные ограничения на имена атрибутов не распространяются на атрибуты символов из правых частей правил вывода грамматики.

После того как атрибуты в левых частях правил и в описании их типов переименованы, для упрощения или исключения некоторых правил вычисления атрибутов можно использовать новое соглашение об обозначениях атрибутов, которое формулируется следующим образом: "Если два атрибута получают одно и то же значение, то им можно дать одно и то же имя при условии, что для этого не нужно изменять имена атрибутов нетерминала в левой части правила вывода".

Рассмотрим несколько примеров.

$A \rightarrow B_x\ C_y\ D_z$

y, z←x

Атрибутам x, y и z присваивается одно и то же значение, поэтому им можно дать общее имя. Используя новое имя a, получим правило вывода грамматики ($A \rightarrow B_x\ C_y\ D_z$), которое не требует правила вычисления атрибута x.

$A_x \rightarrow B_y\ \{f)_z$

y, z←x

Атрибутам можно дать одно имя, но оно должно быть x, для того чтобы не изменилось имя атрибута в левой части правила. Выполнив замену имен, получим правило вывода грамматики $A_x \rightarrow B_x\ \{f\}^*$, при этом отпадает необходимость в атрибутном правиле.

$A_{x,\ y} \rightarrow a, B_z\ C_t$

y, z, t ←x

Атрибуты y, z, t, x имеют одно и то же значение, но им нельзя дать одно имя, поскольку нетерминал в левой части правила вывода имеет два атрибута: x и y. В данном случае можно получить лишь частичное упрощение:

$A_{x,\ y} \rightarrow a, B_y\ C_y$

y←x

Новый способ записи имен атрибутов позволяет непосредственно превращать списки атрибутов нетерминальных символов грамматики в списки параметров процедур для распознавания нетерминальных символов. Такой способ записи имеет недостаток, заключающийся в том, что из него не видно, каким образом между атрибутами передается информация. Например, из правила $A \rightarrow B\ C_x\ D_x$ не ясно, присваивается ли атрибут нетерминала C атрибуту нетерминала D или наоборот. Если атрибут нетерминала C присваивается атрибуту нетерминала Д, то атрибутное правило является L-атрибутным и

может использоваться при реализации перевода методом рекурсивного спуска. В противном случае метод рекурсивного спуска неприменим.

Обратившись к описанию типов атрибутов, можно определить порядок передачи информации между атрибутами (значение синтезированного атрибута должно присваиваться унаследованному атрибуту). Однако на практике более удобно использовать обычный способ именования атрибутов и переходить к новому способу записи только после того, как будет доказано, что исходная АТ-грамматика является L-атрибутной.

Для метода рекурсивного спуска не требуется, чтобы АТ-грамматика, описывающая перевод, имела форму простого присваивания.

Опишем детально, как необходимо расширить метод рекурсивного спуска, чтобы он выполнял атрибутный перевод.

Во-первых, изменим процедуру NextSymbol таким образом, чтобы она читала очередной символ входной цепочки (лексему) и присваивала класс текущего входного символа переменной ClassSymb, а значение лексемы (если оно есть) — переменной ValSymb.

Правила составления процедур при условии, что:

- АТ-грамматика, описывающая перевод, является L-атрибутной;
- левые части правил и описания нетерминалов используют одни и те же имена атрибутов;
- для атрибутов, имеющих одно и то же значение, можно использовать одинаковые имена;
- дополняются следующими правилами:
- формальные параметры. Список имен атрибутов, соответствующий вхождениям нетерминала в левые части правил вывода, становится списком формальных параметров соответствующей процедуры;
- спецификации параметров. Спецификации атрибутов (УНАСЛЕДОВАННЫЙ или синтезированный) переводятся в спецификации формальных параметров по следующим правилам:
- тип унаследованный соответствует способу передачи параметров "вызов по значению";
- тип СИНТЕЗИРОВАННЫЙ соответствует способу передачи параметров "вызов по ссылке";
- локальные переменные. Все имена атрибутов символов данного правила грамматики, кроме тех, что связаны с символом из левой части, становятся локальными переменными соответствующей процедуры; обработка нетерминала из правой части правила. Для каждого вызова процедуры, соответствующего вхождению нетерминального символа в правую часть правила вывода, список атрибутов этого вхождения используется в качестве списка фактических параметров;
- обработка входного символа.

Для каждого вхождения входного символа в правую часть правила вывода грамматики перед вызовом процедуры NextSymb в процедуру включается фрагмент кода, который каждой переменной из списка атрибутов входного символа присваивает значение входного атрибута из переменной ValSymb;

1. обработка операционного символа. Для каждого вхождения операционного символа в правую часть правила вывода грамматики в процедуру включается фрагмент кода, который по соответствующим атрибутным правилам вычисляет значения синтезированных атрибутов операционного символа и присваивает вычисленные значения переменным, соответствующим синтезированным атрибутам. Затем вызывается процедура выдачи операционного символа вместе с атрибутами в выходную строку;
2. обработка правил вычисления атрибутов. Для каждого правила вычисления атрибутов, сопоставленного правилу вывода грамматики, в процедуру включается фрагмент кода, который вычисляет значение атрибута и присваивает это значение каждой переменной из левой части атрибутного правила (если соглашение об одинаковых именах выполнено, то в левой части атрибутного правила будет только один атрибут). Фрагмент кода можно поместить в любом месте процедуры, которое находится:
   - после точки, где используемые в правиле атрибуты уже вычислены;
   - перед точкой, где впервые используется вычисленное значение атрибута;
   - головной модуль. Все имена синтезированных атрибутов начального символа грамматики становятся локальными переменными головного модуля. Список фактических параметров вызова процедуры распознавания начального символа грамматики содержит начальные значения унаследованных атрибутов и имена синтезированных атрибутов.

Пример 1.

В качестве примера реализации атрибутного перевода с использованием метода рекурсивного спуска рассмотрим L-атрибутную транслирующую грамматику.

Для повышения наглядности переименуем атрибуты символов грамматики таким образом, чтобы всем вхождениям символов в правые части разных правил вывода соответствовали разные имена атрибутов, а также выберем одинаковые имена для атрибутов нетерминала R из левой части правил вывода с номерами (3), (4) и (5). Пусть p — унаследованный атрибут нетерминала R, а t — его синтезированный атрибут. После преобразования получим следующую грамматику:

Et синтезированный t

Rp, t унаследованный p синтезированный t

Атрибуты операционных символов унаследованные.

(0) $S_0 \rightarrow S \perp$

(1) $S \rightarrow I_a := E_b$ {:=}a, b

(2) $E_c \rightarrow I_d\ R_{d,c}$

(3) $R_{p,\ t} \rightarrow I_{q1}$ {+}p, q_1, $r_1\ R_{r1,\ t}$

r1 ← GETNEW

(4) $R_{p,\ t} \rightarrow X\ I_{q2}$ {*}p, q_1, , $r_1\ R_{r1,\ t}$

r2 ← GETNEW

(5) $R_{p,\ t} \rightarrow \varepsilon$

t←p

С учетом выполненных преобразований процедура на языке Pascal, реали-

зующая атрибутный перевод операторов присваивания некоторого языка программирования в цепочку тетрад с кодами операций: СЛОЖИТЬ, УМНОЖИТЬ, ПРИСВОИТЬ методом рекурсивного спуска, приведена в листинге 2.2.

Листинг 2.2

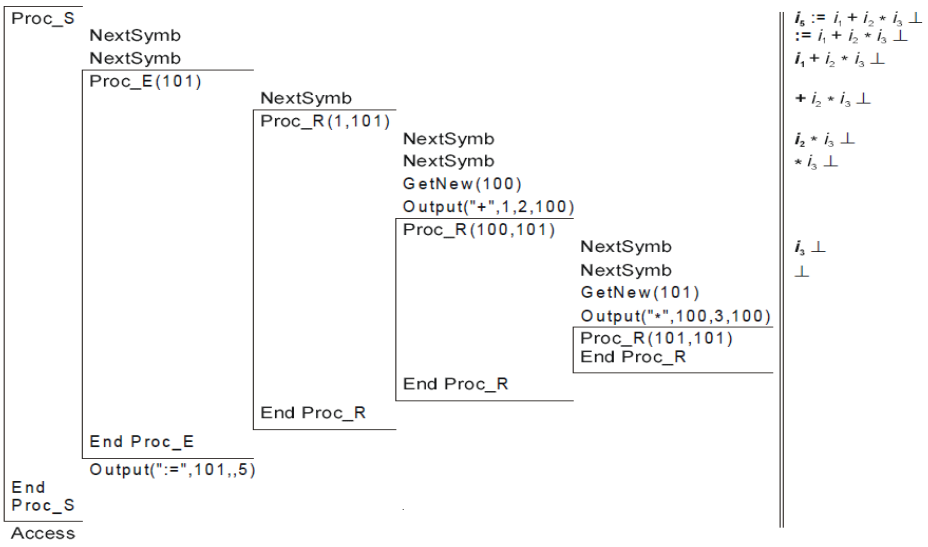
```
Procedure Recurs_Method_ATG (List_Token: tList, List_Tetr: tListTetr);
{List_Token -входная цепочка, List_Tetr - цепочка тетрад}

Procedure Proc_S;
  begin
  if ClassSymb = ClassId {текущий символ - идентификатор} then
  begin
  a := ValSymb;
  NextSymb;
  if ClassSymb = ':=' then
  begin
  NextSymb;
  Proc_E(b);
  Output({:=}, a, b)
  end
  else
  Error
  end
  else
  Error
  end; {Proc_S}
  Procedure Proc_E (var c: integer);
  var d: integer; {локальная переменная Proc_E}
  begin
  if ClassSymb = ClassId {текущий символ - идентификатор} then
  begin
  d := ValSymb;
  NextSymb;
  Proc_R(d,c)
  end
  else
  Error
  end; {Proc_E}
  Procedure Proc_R (p: integer, var t: integer);
  Var q1, q2, r1, r2: integer; {локальные переменные Proc_R}
  begin
     if ClassSymb = '+' then
    begin
  NextSymb;
  if ClassSymb = Classid (текущий символ – идентификатор)  then
  begin
  q2 := ValSymb;
```

```
NextSymb;
GetNew(r2);
Output({+}, p, q2, r2);
Proc_R (r2, t)
end
else
Error;
end
else
t  := p
end; {Proc_R}
begin
if ClassSymb = '*' then begin
{ В начале анализа переменная ClassSymb содержит класс первого символа входной цепочки, а переменная ValSymb - значение этого символа}
Proc_S;
if ClassSymb = '⊥' then
  Access
else
   Error
end {Recurs_Method_ATG};
```

На рис. 1.3 приведен список имен процедур со значениями фактических параметров в порядке их вызова при переводе входной цепочки $i_5 = i_1 + i_2 * i_3 \perp$ в цепочку тетрад. Справа в строке, соответствующей вызову процедуры NextSymb, изображена непрочитанная часть входной цепочки (текущий символ находится слева) непосредственно после вызова этой процедуры.

S-атрибутный ДМП-процессор



Математическая модель восходящего ДМП-процессор

Для любого восходящего синтаксического анализатора, рассматриваемого в данном учебнике, последовательность операций переноса и свертки, выполняемых при обработке допустимых входных цепочек, можно описать с по-

мощью транслирующей грамматики. Входной для этой транслирующей грамматики является грамматика, на основе которой построен анализатор. Для получения транслирующей грамматики в самую крайнюю правую позицию каждого i-го правила входной грамматики вставляется операционный символ {СВЕРТКА, i}. Например, транслирующая грамматика, построенная по входной грамматике G_q = ({E, T\ P), {i, +, *, (,)}, P, E), где P = {E → E + T, E→T, T→ T* P, P, P →i, P → (E)}, будет выглядеть следующим образом:

E → E + T{СВЕРТКА, 1}
E →T{СВЕРТКА, 2}
T→ T* P {СВЕРТКА, 3}
T→ P {СВЕРТКА, 4}
P → i {СВЕРТКА, 5}
P → (E) {СВЕРТКА, 6}

Если каждый входной символ в активной цепочке интерпретировать как представление операции ПЕРЕНОС, выполняемой в момент времени, когда этот символ является текущим входным символом, то активная цепочка в точности описывает последовательность операций переноса и свертки, выполняемых при обработке входной цепочки. Это объясняется тем, что каждая операция свертки выполняется сразу же после того, как локализована соответствующая основа (или первичная фраза), т. е. когда завершается обработка последнего символа в правой части правила вывода. Например, для входной цепочки i + i * i, разбор которой рассматривался в разд. 9.4, активная цепочка, порождаемая рассмотренной ранее транслирующей грамматикой, имеет вид:

i {СВЕРТКА_ 6} + i {СВЕРТКА_6} * i {С В ЕР ТКА_6} {С В ЕРТКА 3} {С В ЕР ТКА_ 1},

что полностью соответствует последовательности операций переноса и свертки, выполняемых при обработке входной цепочки.

Восходящий анализатор можно расширить действиями по выполнению перевода, если перевод определяется постфиксной транслирующей грамматикой. Модификация анализатора заключается в том, что операция свертки расширяется действиями, определяемыми операционными символами соответствующего правила грамматики. Это можно сделать, т. к. при обработке входной цепочки момент времени выполнения свертки для каждого правила грамматики совпадает с моментом выполнения действий по переводу для этого правила. Например, для постфиксной транслирующей грамматики цепочечного перевода:

E → E+ T{+}
E → T
T → T*P {*
T → P
P → i{i}
P → (E)

операции свертки расширяются следующим образом: СВЕРТКА_1 будет обеспечивать выдачу операционного символа {+} в выходную строку,

СВЕРТКА 3 — выдачу операционного символа {*}, а СВЕРТКА 6 — выдачу операционного символа {i}.

Синтаксический анализатор, дополненный формальными действиями по выполнению перевода, принято называть восходящим ДМП-процессором.

Реализация S-атрибутного ДМП-процессора

Восходящий ДМП-процессор можно легко преобразовать в S-атрибутный ДМП-процессор, реализующий атрибутный перевод, определяемый постфиксной S-атрибутной транслирующей грамматикой.

В S-атрибутном ДМП-процессоре каждый магазинный символ имеет конечное множество полей для представления атрибутов. Так же, как и в L-атрибутном ДМП-процессоре, примем, что магазинный символ с п атрибутами представляется в магазине (п + 1)-ой ячейками, верхняя из которых содержит имя символа, а остальные — поля для атрибутов. Поля магазинного символа, предназначенные для атрибутов, заполняются значениями атрибутов в момент вталкивания символа в магазин и не изменяются до момента выталкивания его из магазина.

В S-атрибутном ДМП-процессоре операция переноса расширяется таким образом, что значения атрибутов переносимого входного символа помещаются в соответствующие поля вталкиваемого при переносе магазинного символа.

При выполнении операции свертки для правила с номером i верхние символы магазина представляют собой правую часть i-го правила вывода входной грамматики, а поля магазинных символов содержат значения атрибутов соответствующих символов грамматики.

Расширенная операция свертки использует эти значения для вычисления значений всех атрибутов операционных символов, связанных с правилом вывода транслирующей грамматики, и значений всех атрибутов нетерминала из левой части правила. Значения атрибутов операционных символов используются для выдачи результатов в выходную ленту или выполнения других действий, определяемых этими символами. Атрибуты нетерминала из левой части правила вывода записываются в соответствующие поля магазинного символа, который соответствует этому нетерминалу и вталкивается в магазин во время свертки.

На рис. 6.6 изображена последовательность состояний магазина S-атрибутного ДМП-процессора, осуществляющего перевод Цепочки $i_2 + i_3 \times i_5$ в последовательность тетради со знаками операций СЛОЖИТЬ и УМНОЖИТЬ. S- атрибутная транслирующая Грамматика, входной грамматикой которой является основная грамматика, имеет вид:

E_p- СИНТЕЗИРОВАННЫЙ p

(1)$\mathrm{E}_{p_2} \rightarrow \mathrm{E}_{q_1} + \mathrm{E}_{t_1} \{+\}_{q_2,\ t_2,\ p_1}$

$q_2 \leftarrow q_1$

$t_2 \leftarrow t_1$

$p_1, p_2 \leftarrow$ GETNEW

(3)$\mathrm{E}_{p_2} \rightarrow \mathrm{E}_{q_1} * \mathrm{E}_{t_1} \{*\}_{q_2,\ t_2,\ p_1}$

$q_2 \leftarrow q_1$
$t_2 \leftarrow t_1$
$p_1, p_2 \leftarrow$ GETNEW
(5)$E_{p_2} \rightarrow p_1$
$p_2 \leftarrow p_1$
(6)$E_{p_2} \rightarrow (E_{p_1})$
$p_2 \leftarrow p_1$

На рис. 6.6, а приведено начальное состояние магазина. Рис. 6.6, б представляет результат операции ПЕРЕНОС входного символа с атрибутом 2 в магазин, а рис. 6.6, в результат выполнения операции СВЕРТКА, выполняемой в соответствии с правилом (5). Рис. 6.6, г и д иллюстрируют результат выполнения операции ПЕРЕНОС символов '+' и i_3 соответственно из входной строки в магазин, а рис. 6.6, е - результат выполнения операции СВЕРТКА символа i_3 в E_3. Аналогично выполняются операции переноса и свёртки для символов '*' и i_5 (рис. 6.6, ж-е).

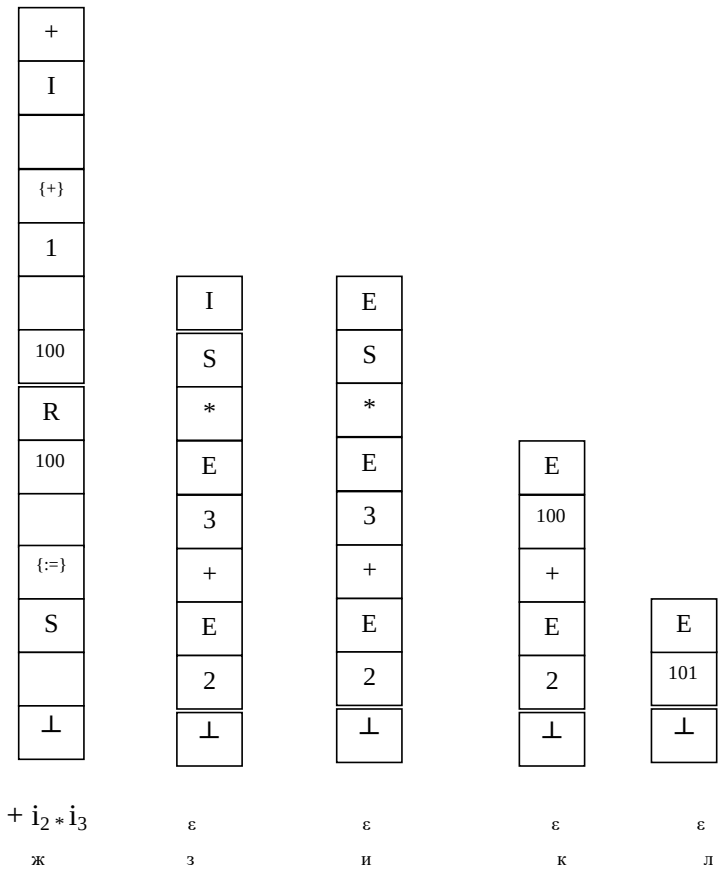


Рис. 6.6. Последовательность состояний магазина при обработке входной цепочки $i_2 + i_3 * i_5$

Рис. 6.6, и иллюстрирует результат выполнения операции СВЕРТКА, выполняемой в соответствии с правилом (1). При этом поля атрибутных

символов из правой Части этого правила используются для формирования тетрады,

*	3	5	100

которая выдаётся в выходную цепочку.

Далее выполняется операция СВЕРТКА по правилу (3) с выдачей тетрады:

+	2	100	101

(рис. 6.6, к-л).

Контрольные вопросы

1. Дайте определение L-атрибутной и S-атрибутной транслирующих грамматик.
2. Какое правило вычисления атрибутов называется копирующим?
3. В каком случае АТ-грамматика имеет форму простого присваивания? Приведите процедуру преобразования произвольной АТ-грамматики в форму простого присваивания.
4. Приведите алгоритм построения управляющей таблицы для транслирующей грамматики Цепочечного перевода, входной грамматикой которой является LL(!)-Грамматика.
5. Опишите, каким образом представляются в магазине и вычисляются унаследованные и синтезированные атрибуты символов Грамматики в L-атрибутном ДМП-процессоре.
6. Приведите процедуру преобразования ДМП-процессора в L-атрибутный ДМП-процессор.
7. Как программируются процедуры в методе рекурсивного спуска, реализующего перевод, описываемый транслирующей грамматикой?
8. С какой целью осуществляется переименование имён атрибутов нетерминальных символов из левых частей правил вывода L-атрибутной транслирующей грамматики при реализации вывода L-атрибутного перевода методом рекурсивного спуска?
9. Как программируются процедуры в методе рекурсивного спуска, реализующего перевод, описываемый L-атрибутной транслирующей Грамматикой?
10. Каким образом восходящий анализатор можно расширить действиями по выполнению перевода?
11. Опишите, каким образом представляются в магазине и вычисляются унаследованные и синтезированные атрибуты символов грамматики в S-атрибутном ДМП-процессоре.

Упражнения

Следующие правила вывода грамматики являются частью некоторой атрибутной грамматики. Атрибуты p, q и r — унаследованные, а s и f — синтезированные. Для каждого правила определите, от каких атрибутов могут зависеть правила вычисления p и r, чтобы это правило было L- атрибутным?

1.1. $A_s, q \rightarrow a_и A_t, p\ B_r$.

1.2. $A_s, q \rightarrow \{c\}p$ Ьи A_t, r.

1.3. $A_s, q \rightarrow$ си $B_r A_t, p$.

Приведите следующие правила вывода АТ-Грамматики к форме простого присваивания (имена унаследованных атрибутов начинаются с символа i, а имена синтезированных атрибутов – с символа s):

2.1. $A_{s_1}, i_1 \rightarrow E$

$s_1 \leftarrow \sin(i_1)$

2.2. $E_{s_1}, i_1 \rightarrow A_{s_2}\ B_{s_3}, i_2\ C_{s_4}, i_3\ D_{s_5}, i_4$

$i_2 \leftarrow i_1$

$i_3 \leftarrow i_1 * i_1$

$i_4 \leftarrow i_2 * i_3$

$s_1 \leftarrow s_2 * s_3$

Преобразуйте L-атрибутную грамматику цепочечного перевода к форме простого присваивания и постройте для нее L-атрибутный ДМП- процессор.

A_p СИНТЕЗИРОВАННЫЙ p

B_p СИНТЕЗИРОВАННЫЙ p

$\{b\}_p$ УНАСЛЕДОВАННЫЙ p

(1) $S \rightarrow a_p\ A_q\ B_r A_s\ \{b\}t$

(3) $A_p \rightarrow$ Ь $r \leftarrow p + 2q$

$p \leftarrow$ Ь

$t \leftarrow r + s$

(2) $A_p \rightarrow a_q\ A_r\ B_s\ B_t B_u S$

(4) $B_p \rightarrow a_q A_r \{b\}_t$

$s \leftarrow q + r$

$t \leftarrow r + q$

$t \leftarrow q * s - 4$

$u, p \leftarrow r + t$

Для Грамматики, заданной в упр. 2, постройте процессор методом рекурсивного спуска.

Для входной грамматики, описывающей синтаксис арифметических выражений, с правилами вывода

$E \rightarrow (E + E)$

$E \rightarrow (E * E)$

$E \rightarrow i$

Где i – лексема, значением которой является указатель на элемент таблицы идентификаторов, определите постфиксную S-атрибутную грамматику, описывающую перевод этих выражений в цепочку тетрад с кодами операций: СЛОЖИТЬ и УМНОЖИТЬ, и постройте S-атрибутный ДМП-процессор, выполняющий заданный перевод.

Приложение А. Порядок выполнения лабораторных работ

Практические занятия по курсу состоят из тематических работ, включающих одну или несколько лабораторных работ.

Лабораторная работа №1-3

Спроектировать автоматную грамматику по заданному языку L, построить конечный автомат.

1. Изучить классификацию Хомского (см. раздел 1.1., 1.2., 1.3.). Ответьте на вопрос, какие грамматики называются автоматными. Какие есть виды автоматных грамматик.

2. Спроектировать по заданному языку L автоматную грамматику и конечный автомат. Используйте пример, и последовательность выполнения работы из раздела 2.3. “Практическая работа 1”.

1. Постановка задачи.
2. Входные и выходные данные.
3.Спроектировать грамматику (Лаб 1).
4.Определить свойства грамматики.
5.Спроектировать конечный автомат , составить диаграмму переходов КА и реализовать на C# (Лаб 2.).

5.1. Создать проект – консольное приложение:

5.2. Исапользуйте следующие франменты программ для реализации КА, распознающего заданный язык.

Пример 1. Создайте проекта простого консольного приложения.

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Linq;
using System.Text;

namespace ConsoleApplication {
   class Program     {
      static void Main(string[] args) {
         ArrayList Q = new ArrayList();
         string str;
         Console.WriteLine("Hello");
         str = Console.ReadLine();
         Console.WriteLine("is= "+str);
         Console.ReadLine();
      }
   }
}
```

Пример 2. Разбор входной строки по символам.

Пример 3. Проектирование и реализация правил.

```
            Delta delta = null;
        // 1. transition
            delta = new Delta("S0", "0", new ArrayList() {"A","qf"});
            DeltaList.Add(delta);
         // 2. transition
            delta = new Delta("A", "1", new ArrayList() { "B"});
            DeltaList.Add(delta);
         // 3. transition
           delta = new Delta("B", "0", new ArrayList() { "A","qf"});
           DeltaList.Add(delta);

        class Delta  {   // структура Delta правил переписывания
           private string LeftNoTerm = null;
           private string LeftTerm      = null;
           private ArrayList Right      = null;
           public string leftNoTerm { get { return LeftNoTerm; } set { LeftNoTerm =
value; } }
           public string leftTerm { get { return LeftTerm; } set { LeftTerm = value; } }
           public ArrayList right { get { return Right; } set { Right = value; } }

            // модель правила
           // delta(  A,                1)            =      {qf}
           //          LeftNoTerm    LeftTerm          Right
           public Delta(string LeftNoTerm, string LeftTerm, ArrayList Right) {
              this.LeftNoTerm = LeftNoTerm;
              this.LeftTerm = LeftTerm;
              this.Right = Right;
           }

           public void DebugDeltaRight() {
              int i = 0;
              for (; i < this.right.Count; i++) {
                 if (i == 0)
                    System.Console.Write(" (" + this.right[i]);
                 else
                    System.Console.Write("," + this.right[i]);
              }
              if (i == 0)
                 System.Console.WriteLine();
              else
                 System.Console.WriteLine(") ");
           }
        } // end class Delta
```

Пример 4. Определение автомата как объекта и тестирование.

```
class Automate { // NDKA (Q,Sigma,deltaList,q0,F)
      ArrayList Q        =  null;        // множество всех состояний
      ArrayList Sigma  =  null;        // конечный алфавит входных символов
      ArrayList DeltaList = new ArrayList();   // множество всех правил
      string    Q0         = null;        // начальное состояние
      ArrayList F          = null;       // заключительные состояния
      // атрибутное программирование на C#
      public ArrayList q { get { return Q; } set { Q = value; } }
      public ArrayList sigma { get { return Sigma; } set { Sigma = value; } }
      public ArrayList deltaList { get { return DeltaList; }
                                   set {DeltaList = value; } }
      public string q0 { get { return Q0; } set { Q0 = value; } }
      public ArrayList f { get { return F; } set { F = value; } }

      public Automate() {
         this.Q           = new ArrayList();
         this.Sigma      = new ArrayList();
         this.DeltaList = new ArrayList();
         this.F           = new ArrayList();
      }

      public Automate(string aname) {
        System.Console.WriteLine(aname);
        // сделать диалог, инициализирующий NDKA
        // альтернативные состояния переходов хранить в массиве см. Test
        //init();
        Test();
      }

      public void Test(){ // задание правил для тестирования
         Q =  new ArrayList() { "S0", "A", "B", "qf" };  // "C"  для отладки
         Sigma  =  new ArrayList() { "0", "1"};
         q0 = "S0";
         F = new ArrayList() { "qf" };
         ... задать правила
      } // end test
```

Пример 5. Считывание символа и выбор правила**:**

```
        foreach (Delta d in this.DeltaList)  {
           if (d.leftNoTerm == q && d.leftTerm == chain.Substring(i,1) ) {
              …
    }
  …
}
```

6.Определить свойства КА. Реализовать алгоритм преобразования НДКА в ДКА (Лаб 3.).

Пример 6. Построение побитового кода булеана для множества мощности n:

Для построения Булеана вначале строим все побитовые комбинации от 0 до $2^n - 1$, где n – мощность множества. Затем побитовая комбинация преобразуется в символы. Например, бинарная последовательность для множества для множества из символов {SAB}, n = 3 приведена в таблице. "1" означает, что символ в подмножестве, “0” – его отсутствие:

Пример 7. Построение по побитовому коду дельта правила:

Реализация алгоритма преобразования на C#:

```
//  NDKA (Q,S,Delta,q0,F)
namespace NDKA2DKA {
   class Program {
      static void Main(string[] args) {
         Automate NDKA = new Automate("NDKA");
         NDKA.Debug();
         Converter converter = new Converter();
         Automate DKA = converter.convert(NDKA);
         DKA.Debug();
         DKA.recognize(DKA.q0, "01010", 0);
      }
   }

   //
   class Converter {
      Automate DKA = null;
      //  множество всех правил deltaListAll
      ArrayList deltaListAll = null;  // all transitions
      //  подмножества, которые содерджат  все заключительные
      //состояния qf, то есть F'
      ArrayList FAll = null;

      public Converter() {}

      public Automate convert(Automate NDKA) {
         // инициализировать данные для каждого вызова convert
         this.DKA = new Automate();
         this.deltaListAll = new ArrayList();
         this.FAll = new ArrayList();

         // Шаг 1. Init q0 & sigma
         DKA.q0      =  NDKA.q0;
         DKA.sigma  =  NDKA.sigma;

         //2. Создать множество всех подмножеств по Q (булеан) и
         //3. Создать множество всех правил DeltaList
```

```
            // 4 и 5,  подмножества, которые содерджат заключительные
            // состояния qf, то есть F'
            BuildDeltaList(NDKA);
            // Шаг 6. Опрелеоить достижимые состояния,
            // исключить недостижимые состояния из множества Q'
            // 1. Берем начальное состояние и определяем правило дельта,
            Reachability(DKA.q0);
            return DKA;
        }

        void BuildDeltaList(Automate NDKA) {
            ArrayList right = null; // для нового правила
            int count =  NDKA.q.Count;
            //    Time Complexity: O(n2^n),   Space Complexity: O(1)
            // 1.  set size of power set of a set with set size  n is (2**n )
            int sizeOfPowerSet = (int)Math.Pow(2, count);
            string leftNoTerm = null; // is subset
            string[] noTerm = null;     // для split
            // 2. Run from counter 000..0 to 111..1
Console.WriteLine("Boolean_____ ");
            for (int counter = 0; counter < sizeOfPowerSet; counter++) {
                leftNoTerm = null;
                for (int j = 0; j < count; j++) {
                    // Check if j-th bit in the counter is set If set then  build set, use comma
// Console.WriteLine("!  0x{0:x8}", 1 << j);
                    if ((counter & 1 << j) != 0) {
// System.Console.WriteLine("NDKA.q[j] = " + NDKA.q[j]);
                    if (leftNoTerm != null)  leftNoTerm = leftNoTerm + ',' + NDKA.q[j];
                        else leftNoTerm = "" + NDKA.q[j];
                    }
                }
                if (leftNoTerm != null) { // 2**n -1 без пустых подмножеств
                    // Найти delta'(S, a)
                   noTerm = leftNoTerm.Split(',');
                   // Шаг 4. построить subset  F
                   BuildFAll(leftNoTerm, NDKA.f);
                   foreach (string leftTerm in NDKA.sigma){
    // по deltaListNDKA посмотреть имеющиеся правила для данного, одно
                       foreach (string n in noTerm) { // ищем правило
                          right = findTransition(n, leftTerm, NDKA.deltaList);
                          if (right != null) {
    deltaListAll.Add(new Delta(leftNoTerm, leftTerm, new ArrayList(right)));
                             break;
                          }
                       }
                    }
System.Console.WriteLine("       "+leftNoTerm); // булеан
```

```
            }
          } // end for
DebugDeltaList(deltaListAll);
DebugF(FAll);
      } // BuildDeltaList

      // найти переход
      public ArrayList findTransition  (string leftNoTerm, string leftTerm,
                                              ArrayList NDKAdeltaList) {
         foreach (Delta d in NDKAdeltaList) { // найдено правило в ArrayList
           if (d.leftNoTerm == leftNoTerm && d.leftTerm == leftTerm)
             return d.right;
         }
         return null;
       }

      void BuildFAll(string leftNoTerm, ArrayList qf) {
       // если в подмножестве noTerm есть заключительное состояние NDKA.f
         string[]  noTerm = leftNoTerm.Split(',');
         foreach (string n in noTerm) { // ищем правило
            foreach (string f in qf) {
               if (n == f) {
                  FAll.Add(leftNoTerm);
// System.Console.WriteLine("  FAll.Add =      " + leftNoTerm);
                  return;
               }
            }
         }
      } // end BuildFAll

       void Reachability(string q) {
       // 1. Берем состояние по правилу дельта и определяем следующее дельта
          string right = null;
          foreach (Delta d in deltaListAll) {
             if (d.leftNoTerm == q) {
               // преобразовать в метку подмножество из right
               d.right = markSubset(d.right);
               DKA.deltaList.Add(d);
               DKA.q.Add(q);
                // всегда один элемент, так как  markSubset
               right = d.right[0].ToString();
               break;
             }
          }
          if (right == null) return; // нет достижимых состояний
          if (DKA.q.Contains(right)) { // это состояние уже было,  останов
            //  заключительное состояние  должно быть F'
```

```
                if (FAll.Contains(right))
                // в F' оставить последнее достижимое состояние
                    DKA.f.Add(right);
                else {
                    System.Console.WriteLine(" Reachability error  " + 01);
                    return;
                }
            }
            else Reachability(right);
        } // end Reachability

        ArrayList markSubset(ArrayList right) {
            string r = null;
            foreach (string s in right) {
                if (r != null) r = r + ',' + s;
                else r = s;
            }
            return new ArrayList(){r};
        }

        void DebugF(ArrayList F) {
            System.Console.WriteLine(" F all:_ ");
            for (int i = 0; i < F.Count; i++) {
                System.Console.WriteLine("        "+F[i]);
            }
        }

        public void DebugDeltaList(ArrayList deltaList) {
            System.Console.WriteLine("deltaList all:_ ");
            foreach (Delta d in deltaList) {
                Console.Write("          (" + d.leftNoTerm + ")," + d.leftTerm + " = ");
                d.DebugDeltaRight();
            }
        }
    } // end class Convertor
```

7. Оформить работу согласно указанным шагам.

Лабораторная работа №4-6

Привести заданную КС-грамматику G = (T, V, P, S) к *приведенной* форме.

1. Изучить алгоритмы приведения КС-грамматик к *приведенной* форме. (см. раздел 3.4.). Ответьте на вопрос, какие КС-грамматики называются грамматиками в *приведенной* форме. Лаб. 4 A,B. Лаб. 5 C,Д., Лаб. 6 F,E.

2. Использовать алгоритмы преобразования КС-грамматик..

А).Устранить из грамматики G бесполезные символы. Применить алгоритм 3.3. к грамматике G.

B). Устранить из грамматика G ε–правила, применить алгоритм 3.5.

C). Устранить из КС грамматики G цепные правила, применить алгоритм 3.6.

D).Устранить левую рекурсию в заданной КС-грамматике G_1, порождающей скобочные арифметические выражения. Применить алгоритм 3.7. к грамматике G.

F). Определить в какой форме (Грейбах, Хомского) находится КС-грамматика G'.

E). G'– *приведенная* КС-грамматика.

3. Оформить работу согласно шагам.

Лабораторная работа №7-8

Построить МП-автомат P и расширенный МП-автомат по КС-грамматики G = (T, V, P, S), без левой рекурсии. Написать последовательность тактов автоматов для выделенной цепочки. Определить свойства автоматов. Лаб. 7. -2.1. Лаб. 8. -2.2.

1. Изучить алгоритмы построения МП-автомат P и расширенного МП-автомата по заданной КС-грамматике (см. раздел 3.4.). Ответьте на вопрос, чем отличается МП-автомат P от расширенного МП-автомата.

2. Выполнить построение согласно алгоритмам. Смотрите пример и последовательность выполнения работы из раздела 3.4. Практическая работа 4.

2.1. A). Построить МП-автомат по КС-грамматике G, используя алгоритм 3.8. Для моделирования магазина ипользуйте Stack из библиотеки C#.

```
Stack <string> stack = new Stack <string> ();
stack.Push("c");
stack.Push("d");

Console.WriteLine(" simbol:   " + stack.Pop());
```

B). Определить последовательность тактов МП-автомата для выделенной цепочк.

2.2. A). Построить расширенный МП-автомат, используя алгоритм 3.9.

B). Определить последовательность тактов расширенного МП-автомата при анализе входной выделенной цепочки P.

3. Определить свойства построенный МП-автоматов.

4. Оформить работу согласно шагам.

Лабораторная работа №9-10

Разработать контекстно-свободную грамматику по заданной строке (см. раздел 3.3.). Алгоритм разбора реализуется в виде процедур, каждой, из которой соответствует диаграмма.

1. Повторить классификацию Хомского (см. раздел 1.1.). Ответьте на вопрос, какие грамматики называются контекстно-свободными. Какие есть виды контекстно-свободных грамматик.

2.Спроектировать по заданной строке контекстно-свободную грамматику и автомат с магазинной памятью. Используйте пример и последовательность выполнения работы из раздела 3.3. Обратите внимание, что в качестве неявного стека могут выступать: рекурсивная процедура и древовидная структура объектов. Лаб. 9 1-4. Лаб. 10 5-7.

1. Спроектировать контекстно-свободную грамматику.
2. Записать вывод заданной строки по грамматики.
3. Определить свойства грамматики.
4. Устранить левую рекурсию и записать вывод заданной строки по грамматики.
5. Оптимизировать грамматику метасимволами {…} и записать вывод заданной строки по грамматики.
6. Составить синтаксический граф для грамматики.
7. Преобразовать граф в программу.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

class State { // абстрактный класс с чисто виртуальной функцией
  public:
    virtual void parse(char c)=0;
};
//подкласс для объекта с состоянием правильного разбора
class OK:public State {
  public:
    OK (){}
    virtual void parse(char c){cout << "OK" << endl;}
};
//подкласс для объекта с состоянием не правильного разбора
class ERROR:public State {
  public:
    ERROR(){}
    virtual void parse(char c){cout << "ERROR string"<< endl;}
};
//класс для автомата агрегация по ссылке объектов классов OK,
ERROR
class Automate {
  public:
    static State* state; // полиморфная переменная
    static ERROR* error;
    static OK* ok;

    Automate(string String){

i=0;

this->String = String;
    }

char getNextChar(){// получить следующий символ из строки
   if (i<(int)String.length()){char ch=String[i];i++;return ch;}
      else {return ' ';}
```

```
        //при окончании строки возвращается пробел

}

virtual void parse(){ // начать разбор

while ( (state !=error)&&(state !=ok) ){
          state->parse(getNextChar());
        }
        state->parse(getNextChar());// принцип подстановки

}
    private:
      string String;

int i;
  }; // end class
  // присвоение начальных значений переменным автомата
  State* Automate::state = NULL;
  ERROR* Automate::error = new ERROR;
  OK*    Automate::ok    = new OK;

  // определение подкласса для объекта автомата, анализирующего
  // контекстно-свободную грамматику
  class AutomateCF: public Automate {
    public:

AutomateCF(string String):Automate(String) {c =' ';}

virtual void parse(){// замещение функции parse в объекте
                              // класса Automate

c = getNextChar(); // вызов функции класса автомат
           E();                // вызов метода

state->parse(getNextChar());

}

void T (){  // реализация функции по диаграмме

cout << "step1 T="<< c <<endl;

if ( (c == 'a'||c == 'b'||c == 'c')){


c = getNextChar();


cout << "step2 T="<< c <<endl;

}

if (c == ')') {c=getNextChar();}
         else if (c == '(') {c=getNextChar(); E();}
```

```
      }

      void E (){// реализация функции по диаграмме
        cout << "step0 E="<<c<<endl;
        T(); // вызов функции
        cout << "step1 E="<<c<<endl;
        while ( (c == '+'||c == '-') ){
          c = getNextChar();T(); // вызов функции

cout << "step2 E="<< c <<endl;
        }
        if (c == ' ') {state=ok;}
        else {state=error;}
      }
    private:

char c;
  };
  // программа ввода строки с клавиатуры
  string getString (){
    //string String = "c+d-dkf-n";
    string  String = "a+(b-c)";
    //  string  String = "a+b";
    char c;
    int N = (int)String.length();
    cout << "Enter string "<<N<<" char "<< endl;
    for (int i=0; i < N;i++){
      cin>>c;
      String[i] = c;
    }
    return String;
  }

  int main(){
    //создание объекта автомат
  Automate * a = new AutomateCF(getString());
    //инициализация начальных значений
    a->state = Automate::ok;
    a->parse();  //синтаксический анализ
    return 0;
  }
```

Обратите внимание, что в переменную “c”, записывается следующий символ, который доступен из функций T() и E(). В функциях указаны трассировочные шаги выполнения, которые необходимо сохранить в программе. Состояния OK и ERROR автомата используются для идентификации выполнения разбора.

3. Оформить работу согласно шагам.

Лабораторная работа №11-10

Реализовать контекстно-свободную грамматику, полученную в работе 5 на основе, таблично-управляемой программы грамматического разбора. Изучить контекстно-зависимые грамматики (см. раздел 1.4.).

1.Повторить классификацию Хомского (см. раздел 1.1.). Ответьте на вопрос, какие грамматики называются контекстно – зависимыми, неограниченными. Какие свойства контекстно – зависимых и неограниченных грамматик, в чем их отличие. Чем они отличаются от изученных раннее грамматик ?

2. Разработать объектно-ориентрованную реализацию для таблично-управляемой программы грамматического разбора.

1. Представить граф грамматики в виде структуры данных (см. раздел 3.3.).
2. Классифицировать типы вершин.
3. Выполнить подстановку графов и получить как можно меньшее число графов.
4. Последовательность вершин графа преобразовать в структуру узлов.
5. Реализовать программу разбора по структуре узлов.

В алгоритме разбора, результат каждого шага разбора выводится на экран, чтобы можно было видеть, как происходит разбор.

```
  #include "stdafx.h"
  #using <mscorlib.dll>

  using namespace System;
  using namespace std;
  class State { // абстрактный класс с чисто виртуальной функцией
    public:
      virtual void parse(char c)=0;
  };
  // класс для объектов вершин, агрегация по указателю
  class Node {
    public:
      Node (char c){ // задание символа вершинам

this->alt = NULL;

this->suc = NULL;

this->sym = NULL;

this->c = c;

}
  // метод для соединения вершин

void link (Node *alt, Node *suc, Node *sym){

this->alt = alt;

this->suc = suc;

this->sym = sym;

}

      Node * sym; // sym != NULL
```

```
      Node * suc; // terminal empty !=' ' && sym == NULL
      Node * alt;
      char c;     // empty=' ' && sym == NULL
  };

  //подкласс для объекта с состоянием правильного разбора
  class OK:public State {
    public:
      OK (){}
      virtual void parse(char c){cout << "OK" << endl;}
  };
  //подкласс для объекта с состоянием не правильного разбора
  class ERROR:public State {
    public:
      ERROR(){}
      virtual void parse(char c){cout << "ERROR string"<< endl;}
  };

  //класс для автомата агрегация по ссылке объектов классов OK,
  // ERROR
  class Automate {
    public:
      static State* state; // полиморфная переменная
      static ERROR* error;
      static OK* ok;

Automate(string String){
    i=0;
    this->String = String;
  }
  char getNextChar(){// получить следующий символ из строки
  if (i<(int)String.length()){char ch=String[i]; i++; return ch;}
        else {return ' ';}
        //при окончании строки возвращается пробел
  }

virtual void parse(){// начать разбор
    while ( (state !=error)&&(state !=ok) ){
          state->parse(getNextChar()); // принцип подстановки
        }
        state->parse(getNextChar());
  }
    private:
      string String;

int i;
  }; // end class

  // присвоение начальных значений переменным автомата
  State* Automate::state = NULL;
  ERROR* Automate::error = new ERROR;
  OK*    Automate::ok    = new OK;

  // определение подкласса для объекта автомата, анализирующего
  // контекстно-свободную грамматику
```

```
  class AutomateCF: public Automate {
    public:

AutomateCF(string String, Node &node):Automate(String) {
    this ->node = &node; // передача начальной вершины
      c =' ';
}

void parse(Node * nd, bool b){
    Node *p = nd->alt; // Node head
    do {
  if (p->sym == NULL){ // терминал или пусто “empty”
    if (p->c == c){
  cout<< "step01 c=" << c<< ": p->c ="<<p->c<<endl;
  b=true; c=getNextChar();
  state=ok;
              // конец строки
       if (c == ' ') {cout<< "step012 c="<<endl; return;}
    } else if  (p->c ==' ') { // “empty”
    cout<< "step02 if c=" << c<< ": p->c ="<<p->c<<endl;
    b=true;
    } else { b=false; state=error;

cout<< "step03 c=" << c<< ": p->c ="<<p->c<<endl;
    }
  }
  else if (p->sym != NULL){ //- nil
      cout<< "step04 c=" << c<< "p->sym ="<<p->sym<<endl;
      parse(p->sym, b);
  }
  if (b) {
    cout<< "step05 ok: next c=" << c<< endl;
    p=p->suc;
  }
  else {p=p->alt;}
    } while (p != NULL);
  } // end parse

virtual void parse(){ // замещение функции parse в объекте
        c = getNextChar();
    parse(this->node, true);
    state->parse(getNextChar());
  }
    private:
      Node * node;
  char c;
  };

  // программа ввода строки с клавиатуры
  string getString (){ // различные варианты для тестирования..
    // string String = "c+d-dkf-n";
    // string  String = "a+(b-cba)-c";
    // string  String = "a+b";
    // string  String = "a+b-c-c+a";
    // string  String = "a+gfg+fgfg"; // error
```

```
    string  String = "a+(b-c)";
  char c;
  int N = (int)String.length();
  cout << "Enter string "<<N<<" char "<< endl;
  for (int i=0; i < N;i++){
    cin>>c;
    String[i] = c;
  }
  return String;
}

int main(){
// Построение графа
// 1. Объявление вершин
  Node E = Node(' ');
  Node braceL = Node('(');
  Node nil = Node(' ');
  Node braceR = Node(')');
  Node c_a = Node('a');
  Node c_b = Node('b');
  Node c_c = Node('c');
  Node emptyR = Node(' ');
  Node plus = Node('+');
  Node minus = Node('-');
  Node nil_1 = Node(' ');

  // 2. Соединение вершин в дерево методом link
  // void link (Node * alt, Node * suc, Node * sym)
  E.link(&braceL,NULL,NULL);
  braceL.link(&c_a,&nil,NULL);
  nil.link(NULL,&braceR,&E);
  braceR.link(NULL,&plus,NULL);
  c_a.link(&c_b,NULL,NULL);
  c_b.link(&c_c,NULL,NULL);
  c_c.link(&emptyR,NULL,NULL);
  emptyR.link(NULL,&plus,NULL);
  plus.link(&minus,&nil_1,NULL);
  minus.link(NULL,&nil_1,NULL);
  nil_1.link(NULL,NULL,&E);

  //создание объекта автомат
Automate * a = new AutomateCF(getString(),E);
  //инициализация начальных значений
  a->state = Automate::ok;
  a->parse(); //синтаксический анализ

  return 0;
}
```

6. Оформить работу согласно шагам.

Лабораторная работа №12-13.

Построить управляющую таблицу M для LL(k)-грамматики, написать правило вывода, опредилить является ли G грамматика *сильно* LL(k)-грамматикой (см. раздел 3.4.).

1. Изучить раздел 3.4. Построение управляющей таблицы M для грамматики G =(T, V, P, S), работу алгоритма для определенной цепочки и определение *сильно* LL(k)-грамматики.

Пример построения управляющей таблицы.

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Linq;
using System.Text;

namespace lab {
    class Program  {
        static void Main(string[] args) {
           MTable mTable = new MTable();
           // 1 row
           Row row = new Row();
           row.addItem("TE',1");
           row.addItem("TE',1");
           row.addItem("");
           row.addItem("");
           row.addItem("");
           row.addItem("");
           mTable.setRow(row);
           row = new Row();
           row.addItem("");
           row.addItem("");
           row.addItem("E,3");
           row.addItem("+TE',2");
           row.addItem("");
           row.addItem("E,3");
           mTable.setRow(row);
       //      row.DebugRow();
            mTable.DebugTable();
          }
        }
    }//конец для namespace

    class MTable {
        ArrayList M = new ArrayList(); //
        public void setRow(Row row) { M.Add(row); }
        public void DebugTable() {
            //foreach (row r in M) {
               for(int i=0;i<M.Count;i++){
```

```
                    ((Row) M[i]).DebugRow();
                }
                //Console.WriteLine(r);
            }
        }

        class Row {     // строка таблицы любого размера из
string - это колонка
            ArrayList row = new ArrayList(); //
            public void addItem (string item) {
               row.Add(item);
            }
    //        public ArrayList getRow() { return row; }
            public void DebugRow() {
                string[] str = null;
                foreach (string r in row) {
                    Console.WriteLine(r);

                    str = r.Split(',');
                    foreach (string  l in str) {
                        Console.WriteLine(l);
                    }
                }
            }
        }
```

2. Определить размеры управляющей таблица M. В соответствии с шагами алгоритм 3.10 построить управляющую таблицу M (см. раздел 3.3. “Практическая работа 7.”)

3. Выделить цепочку, принадлежащую языку порождаемому грамматикой G и написать правила вывода для цепочки.

4. Для LL(k)-грамматики G определить является ли она *сильно* LL(k)-грамматикой.

5. Оформить работу согласно шагам.

Лабораторная работа №14-15

Построить управляющую таблицу M для LR(k)-грамматики, написать правило вывода выделенной строки (см. раздел 3.4.). Описать работу алгоритма LR(k) анализатора.

1. Изучить раздел 3.4. Построение управляющей таблицы M для LR(k)-грамматики G =(T, V, P, S).

2. В соответствии с шагами алгоритм 3.12 построить управляющую таблицу M (см. раздел 1.4. “Практическая работа 8.”)

3. Выделить цепочку, принадлежащую языку порождаемому грамматикой G и написать правила вывода для цепочки.

4. Для выделенной цепочки показать работу LR(k)-анализатора в соответствии с шагами алгоритм 3.11.

5. Оформить работу согласно указанным пунктам и используемым шагам алгоритмов.

Лабораторная работа №16

Применить алгоритм типа “перенос-свертка” для заданной грамматики G= (T, V, P, S). Описать работу алгоритма.

1. Изучить алгоритм типа “перенос-свертка”
2. В соответствии с шагами алгоритм 3.13 по шагам рассмотреть работу алгоритма (см. раздел 3.4. “Практическая работа 8.”)
3. Выделить цепочку, принадлежащую языку порождаемому грамматикой G и написать правила вывода для цепочки.
4. Для выделенной цепочки показать работу LR(k)-анализатора в соответствии с шагами алгоритм 3.13 .

5. Оформить работу согласно указанным пунктам и используемым шагам алгоритма.

Приложение В. Задания к лабораторным работам

Для лабораторных работ 1-3 определены варианты автоматных языков:

1. $L=\{0\omega_1+(01)^* \mid \omega_1 \in \{0,1\}^*\}$
2. $L=\{01-(10)^*+\omega_1 01 \mid \omega_1 \in \{0,1\}^*\}$
3. $L=\{0(00)^*+01\omega_1 \mid \omega_1 \in (0,1,2)^*\}$
4. $L=\{\omega_1\omega_2 1 \mid \omega_1 \in \{1,0\}^+, \omega_2 \in \{1,0\}^+\}$
5. $L=\{1\omega_1 1-(00)^* \mid \omega_1 \in \{1,0\}^*\}$
6. $L=\{\omega_1-0\omega_2-0+\omega_3 \mid \omega_1 \in \{0,1\}^+, \omega_2 \in \{0,1\}^+, \omega_3 \in \{0,1\}^+\}$
7. $L=\{(0+1)(01)^*+\omega_1 \mid \omega_1 \in \{0,1\}^+\}$
8. $L=\{(0+1)^*\omega_1\omega_2 \mid \omega_1 \in \{0,1,2\}^+, \omega_2 \in \{0,1\}^+\}$
9. $L=\{(01)^*-1-(01)^*+\omega_1 \mid \omega_1 \in \{0,1\}^+\}$
10. $L=\{1(01)^*0-1+\omega_1 \mid \omega_1 \in \{0,1\}^+\}$
11. $L=\{(0+1)+\omega_1+(01)^*0 \mid \omega_1 \in \{0,1\}^+\}$
12. $L=\{0(000)^*(0+1)\omega_1 \mid \omega_1 \in \{0,1\}^*\}$
13. $L=\{1(01)^*(01)\omega_1 \mid \omega_1 \in \{0,1\}^*\}$
14. $L=\{00\omega_1+(1)^* \mid \omega_1 \in \{0,1\}^*\}$
15. $L=\{0(01)^*+1\omega_1 0 \mid \omega_1 \in \{1,0\}^+\}$
16. $L=\{1\omega_1 1\omega_2 1 \mid \omega_1 \in \{0,1\}^+, \omega_2 \in \{0,1\}^+\}$
17. $L=\{011\omega_1 1(0)^* \mid \omega_1 \in (0,1)^+\}$
18. $L=\{\omega_1 0-\omega_2 \mid \omega_1 \in \{0,1\}^+, \omega_2 \in \{0,1\}^+\}$
19. $L=\{\omega_1(1)^*0\omega_2 \mid \omega_1 \in \{1,2\}^*, \omega_2 \in \{0,1\}^*\}$
20. $L=\{10+\omega_1(10)^*\omega_2 \mid \omega_1 \in \{1,2\}^*, \omega_2 \in \{0,1\}^*\}$
21. $L=\{10\omega_1 0-1\omega_2 \mid \omega_1 \in \{0,1\}^+, \omega_2 \in \{1,2\}^*\}$
22. $L=\{1-1\omega_1 01\omega_2 \mid \omega_1 \in \{1,2\}^+, \omega_2 \in \{1,2\}^*\}$

Лабораторные работы 4-6

А).Устранить из грамматики G =(T, V, P, S) бесполезные символы, где

1. $P=\{S \rightarrow b, S \rightarrow F, S \rightarrow cFB, A \rightarrow Ab, A \rightarrow c, B \rightarrow cB, F \rightarrow Ca, C \rightarrow d\}$
2. $P=\{S \rightarrow b, S \rightarrow cAB, A \rightarrow Ab, A \rightarrow c, B \rightarrow cB, C \rightarrow Ca, F \rightarrow d\}$

3. P={S → b, S → BA, A→ Ab, A→ c, B → cB, F→ Ca, C→ d}
4. P={S → cB, B → cB, B → cA, A→ Ab, C→ Ca, F→ d}
5. P={S → c, F → A, S→ cAB, A→ Ab, A→ c, B → cB, C→ Ca, C→ d}
6. P={S → cFCB, A→ ACb, A→ cC, B → cB, C→ Ca, F→ d}
7. P={S → b, S → CF, S→ cCB, A→ Ab, A→ c, B → cB, C→ Ca, F→ d}

B). Устранить из грамматика G=(T, V, P, S) ε–правила, где
1. P={S →AB, A → SA, A→ BB, A→ bB, A→ c, B → c, B→ ε}
2. P={S → b, S→ cAB, A→ Ab, A→ c, B → cB, B→ ε}
3. P={S → bA, S → bA, A→ Ab, A→ ε , B → cB, B→ ε }
4. P={S → cB, B → cB, B → cA, A→ ACb, A→ ε, C→ Ca, C→ ε }
5. P={S → c, S→ cAB, S→ ε, A→ c, B → cB, B → ε}
6. P={S → cFCB, A→ ACb, A→ cC, B → cB, , B → ε, C→ Ca, F→ ε }
7. P={S → b, S → C, S→ cCB, A→ Ab, A→ c, B → cB, C→ Ca, C→ ε}

C). Устранить из КС грамматики G цепные правила, где
1. P={S →AB, A → S, A→ B, A→ bB, A→ c, B → c}
2. P={S → b, S→ cAB, A→ Ab, A→ B, B → cB, B→ b}
3. P={S → bA, S → bA, A→ Ab, A→ S, B → cB, B→ c }
4. P={S → cB, B → cB, B → cA, A→ C, A→ aB, C→ Ca, C→ cf }
5. P={S → c, S→ cAB, S→ a, A→ B, B → cB, B → f}
6. P={S → cFCB, A→ ACb, A→ C, B → cB, , B → b, C→ Ca, F→ c }
7. P={S → b, S → C, S→ cCB, A→ Ab, A→ C, B → cB, C→ Ca, C→ b}

D). Исключить левую рекурсию из КС – грамматики G, где
1. P={S →Bc, S → Ad, A→ Sa, A→ AbB, A→ c, B → Sc, B → b}
2. P={S → FA, S → c, A→ FS, A→ Sa, B → SB, B→ b, F→ f}
3. P={S → Bb, B → Sa, B → cB, B→ Ac, A→ cSB, A→ a}
4. P={S → Ba, S → Ab, A → Sa, A→ AAb, A→ c, B→ Sb, B→ b }
5. P={S → ScB, S→ cAB, S→ c, A→ AbB, B → b, B → aA}
6. P={S → cFCB, A→ ACb, B → cB, , B → b, C→ Ca, , C→ c, F→ f }
7. P={S → AB, S → SC, A→ BB, A→ Ab, A→ a, B → b, C→ Ca, C→ b}

Лабораторные работы 3-8 задана КС-грамматика G =(T, V, P, S), где
1. T = {i, =, *,(,)}, V = {S, F, L}, P = {S→ F = L, S→ L, F→ (* L), F→ i, L→ F}
2. T = {i, &, *,(,)}, V = {S, F, L}, P = {S→ F & L, S→ (S), F→ * L, F→ i, L→F}
3. T = {i, ^ , -,(,)}, V = {S, F, L}, P = {S→ (F ^ L), F→ - L, F→ i, L→F}
4. T = {i, * , -,(,)}, V = {S, F, L}, P = {S→ (F)* L, F→ - L, F→ i, L→F}
5. T = {i, + , -,(,)}, V = {S, F, L}, P = {S→ (F)+(L), F→ - L, F→ i, L→F}
6. T = {i, + , -,(,)}, V = {S, F, L}, P = {S→ (F)+(L), F→ - L, F→ i, L→F}
7. T = {i, + , -,(,)}, V = {S, F, L}, P = {S→ (F+L), S→ (F), F→ - L, F→ i, L→F}
8. T = {i, & , ^,(,)}, V = {S, F, L}, P = {S→ F^L, S→ (F), F→ &L, F→ i, L→F}
9. T = {i, & , ^,(,)}, V = {S, F, L}, P = {S→ F^L, S→ (S), F→ &L, F→ i, L→F}
10. T = {i, & , ^,(,)},V= {S, F, L}, P = {S→ (F^L), S→ (S), F→ &L, F→ i, L→F}
11. T = {i, & , ^,(,)},V={S, F, L}, P = {S→ (F^L), F→ &L, F→ i, L→F}
12. T = {i, & , ^,(,)},V={S, F, L}, P = {S→ &F^, S→ (L), F→ L, F→ i, L→F}
13. T = {i, * , :,(,)}, V={S, F, L}, P = {S→ F: L, S→ (L), F→ L*, F→ i, L→F}

14. T = {i, * , :,(,)}, V={S, F, L}, P = {S→(F: L), F→ L*, F→ i, L→F}
15. T = {i, * , :,(,)}, V={S, F, L}, P = {S→(F: L), S→(F), F→ L*, F→ i, L→F}
16. T = {i, * , +,(,)}, V={S, F, L}, P = {S→(F+ L), F→ L*, F→ i, L→F}
17. T = {i, * , +,(,)}, V={S, F, L}, P = {S→(F+ L), F→ (L*), F→ i, L→F}
18. T = {i, * , +,(,)}, V={S, F, L}, P = {S→F+ L, F→ (L*), F→ i, L→F}
19. T = {i, * , +,(,)}, V={S, F, L}, P = {S→F+ L, S→(S), F→ L*, F→ i, L→F}
20. T = {i, @ , &,(,)},V={S, F, L}, P = {S→(F@L), S→(F&L), F→ i, L→F}
21. T = {i, + , -,(,)}, V={S, F, L}, P = {S→F+L, S→(S), S→(L-), F→ i, L→F}

Л**абораторные работы 14-16**, правила G грамматики рассмотреть как правила LL(k) грамматики, а для работы 7 как правила LR(k).

Заключение

Представленный теоретический материал служит основой для практического проектирования грамматик по заданным языкам и эквивалентным грамматикам автоматов, для реализации синтаксических анализаторов на основе объектно-ориентированного подхода.

Лабораторные работы могут быть использованы для формирования курсовых и дипломных работ, а также исследовательских работ по теме теория автоматов и языков.

Библиографический список

1.*Ахо А., Ульман Д.* Теория синтаксического анализа перевода и компиляции // Пер. с англ. - М.: Мир, 1978.

2.Ахо, Альфред, В., Сети, Рави, Ульман, Джеффри, Д. Компиляторы: Принципы, технологии, инструменты. Пер. с англ. — М.: Издательский дом "Вильяме", 2003.

3.*Р.Хантер.* Проектирование и конструирование компиляторов. Пер. с англ. – М.: Финансы и статистика, 1984.

4.*Н.Вирт.* Алгоритмы + структуры данных = программы. Пер. с англ. – М.: МИР, 1985.

5.*C.Moore*. Dynamical Recognizers: Real-time Language Recognition by Analog Computers. Theoretical Computer Science **201**, 1998, pp. 99-136.

6.*W.Tabor*. Fractal Encoding of Context Free Grammars in Connectionist Networks. University of Connecticut Expert Systems 17(1), 2000, pp. 41-56

7.*А.С.Семенов.* Построение класса фрактальных систем по шаблону на примере дерева Фибоначчи // РАН. Информационные технологии и Вычислительные системы. – М.: N2, 2005 стр.10-17.