А.С. СЕМЕНОВ

ПРАКТИЧЕКИЙ КУРС ПО ТЕОРИИ АВТОМАТОВ И ФОРМАЛЬНЫХ ЯЗЫКОВ

С ПРИМЕРАМИ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C#

Москва 2019

СОДЕРЖАНИЕ

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.

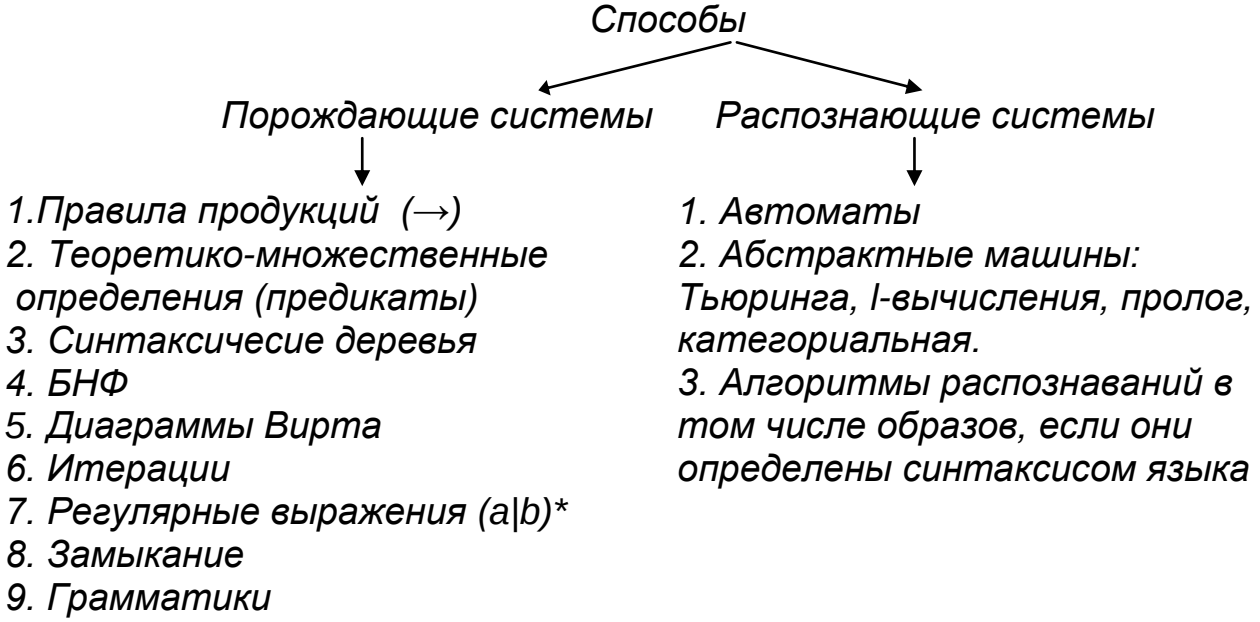


Рис. 1.1. Способы описания синтаксиса языков

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

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

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.

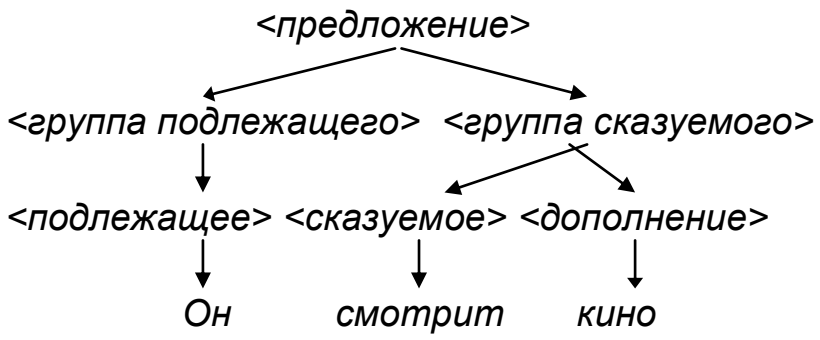


Рис. 1.2. Древовидная структура предложения

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 ξ ::= η 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 α ::= β | $\beta\gamma$ можно представить α ::=β [γ].

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*

Две последовательности связаны отношением $\Rightarrow^*$, когда вторая получается из первой применением некоторой последовательности продукции $\alpha \Rightarrow^* \alpha_m$ тогда, и только тогда, когда $\alpha \Rightarrow^+ \alpha_m$.

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**)

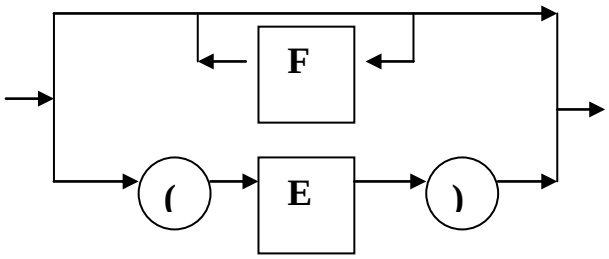


Рис. 1.3. Диаграммы Вирта

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.**

$(a)^* = \{a, aa, aaa, aaaa, \ldots\}$

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:

$$L = \{0^n 1^n \mid n \geq 0\}$$

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 вывода* и записывается в виде $\alpha \rightarrow \beta$.

Цепочка $\beta \subset (T \cup V)^*$ *непосредственно выводима* $(\alpha \to \beta)$ 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 вывода $\gamma \to \delta$ содержится в P.

Цепочка $\beta \subset (T \cup V)^*$ *выводима* из цепочки $\alpha \subset (T \cup V)^+$ в грамматике G, (обозначим $\alpha \Rightarrow \beta$), если существуют цепочки $\gamma_0, \gamma_1, \ldots, \gamma_n$ $(n \geq 0)$, такие, что $\alpha = \gamma_0 \to \gamma_1 \to \ldots \to \gamma_n = \beta$. Последовательность $\gamma_0, \gamma_1, \ldots, \gamma_n$ называется *выводом длины n*.

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

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

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

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.

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) $\alpha \to \beta$, $\mid \alpha \mid \leq \mid \beta \mid$	$S_0 \to 0B0$ $B \to 1 \mid 0BC$ $C0 \to 100$ $C1 \to 1C$	$\{0^n1^n0^n \mid n \geq 1\}$	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	$\{0^n1^n \mid n \geq 1\}$	с магазинной памятью (МП-автомат: LL-разбор, РМП-автомат: LR-разбор) Распознаватели: LL(1) LR(1) подкласс LR(1) грамматики предшествования
3	Регулярная $A \to bB$ $B \to a$	$S_0 \to 0 \mid 0A$ $A \to 0B$ $B \to 1 \mid 1A$	$\{0(01)^n \mid n \geq 0\}$	Конечный автомат (КА)

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.

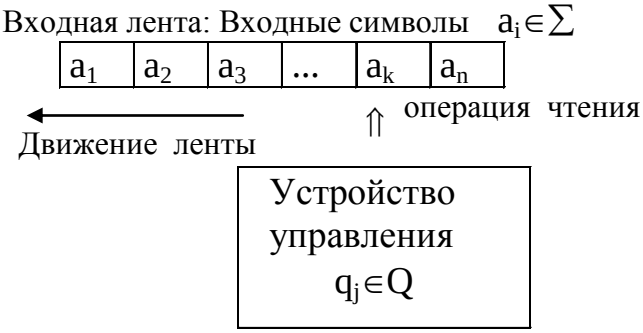


Рис. 2.1. Схема конечного автомата

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

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 $(q, \omega) \in Q \times \Sigma^*$, где $q \in Q$, $\omega \in \Sigma^*$. 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 $\omega \in \Sigma^*$.
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 = $(Q, \Sigma, \delta, q_0, F)$ распознает входную цепочку $\omega \in \Sigma^*$, если $(q_0, \omega) \vdash^k (q, \varepsilon)$ для $q \in F$.

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

$L(КА) = \{\omega \in \Sigma^* | (q_0,\omega) \vdash^k (q, \varepsilon)$, 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 = $(Q, \Sigma, \delta, q_0, F)$, что $L(КА) = L(G)$. КА строится следующим образом:

1. Входные алфавиты конечного автомата Σ и автоматной грамматики T совпадают, $\Sigma = T$.

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:

- $q_f \in \delta(B,a)$, если $B \to a \in P$, $B \in V$, $a \in \sum$;
- если $B \to aC \in P$, то $C \in \delta(B,a)$;
- $\delta(q_f , a) = \varnothing$ 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: $S_0 01010 \vdash A1010 \vdash B010 \vdash A10 \vdash B0 \vdash q_f , \varepsilon$

2.2. Способы задания конечных автоматов

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

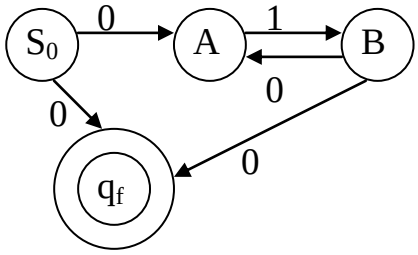


Рис. 2.2. Диаграмма переходов КА

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 слова регулярного языка можно *накачать*, то есть **повторить** внутреннюю часть слова сколько угодно раз, производя новое слово, также принадлежащее языку. Лемма описывает существенное свойство всех регулярных языков и служит инструментом для доказательства нерегулярности некоторых языков. Она является одной из нескольких лемм о накачке.

Два способа:

1. построить распознаватель для заданного языка - конечный автомат;
2. 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).

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

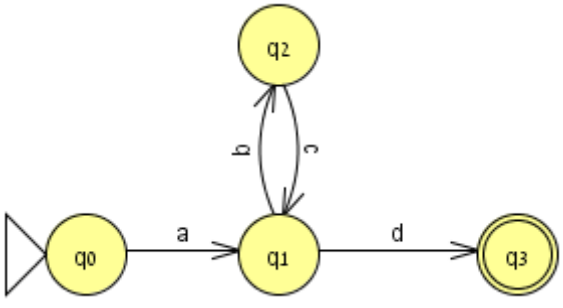


Рис. 2.3. Диаграмма переходов КА

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 $|xy| \leq p$, *где p – длинна накачки*.

Длина накачки p принимается более длины самого длинного слова языка.

Неформальное утверждение: слово w языка L длины по меньшей мере p, (где p константа, называемая длиной накачки, зависит лишь от L) можно разделить на три подцепочки, $w = xyz$, так что среднюю часть, y (непустую), можно повторить произвольное число раз (включая ноль, **то есть удалить**) и получить строку из L. Этот процесс повторения называется «накачкой».

Причем, длина xy не превысит p, ограничивая способы разделения строки w.

Конечные языки удовлетворяют требованиям леммы о накачке тривиально определяя m длиной максимальной строки из языка плюс один.

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.

Формальное утверждение.

Пусть L регулярный язык. Тогда существует целое $p \geq 1$ зависящее только от 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), при этом:

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

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

Пример 3. Нерегулярность языка $L = \{a^nb^n \mid n \geq 0\}$ 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 $w = a^pb^p$. 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.

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.

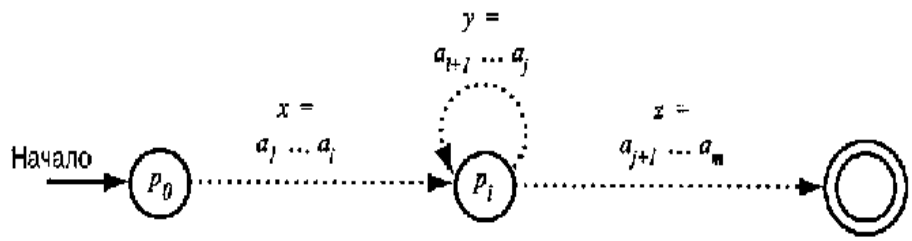
Доказательство. Пусть L – регулярный язык, тогда L = L(ДКА) для некоторого ДКА. Пусть ДКА имеет n состояний. Рассмотрим произвольную цепочку w длинной не менее n. и скажем, $w = a_1a_2a_3..a_m$, где $m \geq n$ и каждый a_i есть входной символ. Для $i = 0,1,2,\ldots,n$ определим состояние p_i как $\delta(q_0,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

Заметим, что $p_0 = q_0$.

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

1. $x = a_1a_2...a_i$
2. $y = a_{i+1}a_{i+2}...a_j$
3. $z = a_{j+1}a_{j+2}...a_m$

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

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. Способы преобразования НКА в ДКА

1.Построение НКА из автоматов.

?????????????????????????????? см. лаб.

2.Детерминированный конечный автомат ДКА = $(Q, \Sigma, \delta, q_0, F)$, если множество $\delta(q, a)$ содержит не более одного состояния для любых $q \in Q$, $a \in \Sigma$. Рассмотрим алгоритм построения ДКА по недетерминированному КА (НКА).

Основная идея построения ДКА по НКА заключается в том, что после обработки отдельной входной цепочки *состояние ДКА будет представлять собой множество всех состояний НКА*, которые он может достичь из начальных состояний после применения данной цепочки.

Переходы ДКА можно получить из НКА, вычисляя множества состояний, которые могут следовать после данного множества при различных входных символах.

Допустимость цепочки определяется исходя из того, является ли последнее детерминированное состояние, которого достиг ДКА, множеством недетерминированных состояний, включающим хотя бы одно допускающее состояние.

Вход: НКА = $(Q, \Sigma, \delta, q_0, F)$

Выход: ДКА = $(Q', \Sigma, \delta', q_0', F')$, у которого L(НКА) = L(ДКА).

1. Входные алфавиты НКА и ДКА совпадают.
2. Множество Q’ содержит множество всех подмножеств множества Q (булеан множества Q). Если $|Q| = n$, то $|Q'| = 2^n$.

3. Для каждого множества $S \subseteq Q$ и каждого входного символа $a \in \sum$, выполняется $\delta'(S, a) = \bigcup_{p \text{ из } S} \delta(p, a)$, т.е. чтобы найти $\delta'(S, a)$ необходимо для каждого состояния p из S найти состояния, в которые можно попасть из p для входного $a \in \sum$. Затем взять объединение множеств всех состояний по всем состояниям p.
4. F' есть множество подмножеств S множества Q, $S \cap F \neq \varnothing$, то есть F' состоит из всех множеств состояний Q НКА, содержащего хотя бы одно заключительное состояние.
5. $q_0' = q_0$.
6. Не все состояния из Q' достижимы из начального состояния (состояние q достижимо из начального состояния q_0', если имеется путь из состояния q_0' в состояние q). Недостижимые состояния надо исключить из множества Q'.
7. Определить ДКА.

Построим ДКА, допускающий язык, определяемый НКА для рассматриваемого примера рис. 2.2.

1. Пусть входные алфавиты НКА и ДКА совпадают.

2. Булеан множества без пустого подмножества $2^4 - 1 = 15$, Q' = {{S}, {A}, {B}, {q_f}, {S,A}, {S,B}, {S,q_f}, {A,B}, {A,q_f}, {B,q_f}, {S,A,B}, {S,A,q_f}, {A,B,q_f}, {S,B,q_f}, {S,A,B,q_f}}

3. Найти $\delta'(S, a)$:

$\delta'(\{B\}, 0) = \{A,q_f\}$ $\delta'(\{S\}, 0) = \{A, q_f\}$ $\delta'(\{A\}, 1) = \{B\}$
$\delta'(\{S,B\}, 0) = \{A,q_f\}$ $\delta'(\{S,A\}, 0) = \{A, q_f\}$ $\delta'(\{S,A\}, 1) = \{B\}$
$\delta'(\{S,q_f\}, 0) = \{A,q_f\}$ $\delta'(\{B,A\}, 0) = \{A,q_f\}$ $\delta'(\{B,A\}, 1) = \{B\}$
$\delta'(\{B,q_f\}, 0) = \{A,q_f\}$ $\delta'(\{A,q_f\}, 1) = \{B\}$ $\delta'(\{S,B,A\}, 1) = \{B\}$
$\delta'(\{S,B,A\}, 0)=\{A,q_f\}$ $\delta'(\{S,B,q_f\}, 0)=\{A,q_f\}$ $\delta'(\{B,A,q_f\}, 0) = \{A,q_f\}$
$\delta'(\{B,A,q_f\}, 1) = \{B\}$ $\delta'(\{S,A,q_f\}, 0) = \{A,q_f\}$ $\delta'(\{S,A,q_f\}, 1) = \{B\}$
$\delta'(\{S,B,A,q_f\}, 0) = \{A,q_f\}$ $\delta'(\{S,B,A,q_f\}, 1) = \{B\}$

4. F' = {{q_f}, {A,q_f}, {S,q_f}, {B,q_f}, {S,B,q_f}, {B,A,q_f}, {S,A,q_f}, {S,B,A,q_f}} Каждое из подмножеств F' является заключительным состоянием. При определении достижимого заключительного состояния выбирается соответствующее подмножество.

5. $S = S_0$

6. Достижимыми состояниями в ДКА являются {S}, {B} и {A,q_f}, остальные состояния удаляются.

Таб. 2.2. Таблица переходов для ДКА

	0	1
p_0 = {S}	p_1	
p_1 ={A,q_f}		p_2
p_2 = {B}	p_1	

7. ДКА = ({{S}, {B}, {A, q_f}}, {0, 1}, δ', S, {{A,q_f }}), где
$\delta'(\{B\}, 0) = \{A,q_f\}$ $\delta'(\{S\}, 0) = \{A, q_f\}$
$\delta'(\{A,q_f\}, 1) = \{B\}$

ДКА представляется следующим графом:

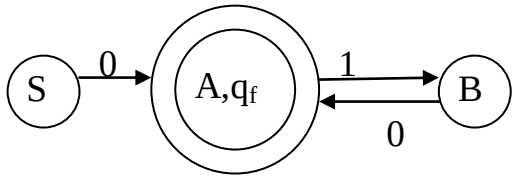


Рис. 2.5. Диаграмма переходов ДКА

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

2.5.Способы реализации: по заданному регулярному выражению составные автоматы

Реализовать на языке C# КА см. рис. 2.5. Пользователь вводит цепочку символов в поле редактирования и получает ответ: “Да” – цепочка символов принадлежит языку, автомат распознал цепочку или “Нет” – цепочка символов не принадлежит заданному языку L.

Пример проектирования грамматики и распознавателя

1. Постановка задачи.

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, ∑, δ, 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. 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.b. 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.c. Используя грамматику G построить КА.

КА= ($\{S_0, Q, q_f\}$, { c, d, k, f, n, +, -}, δ, 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.d. Определить свойства конечного автомата. Конечный автомат является детерминированным.
1.e. Построить диаграмму переходов КА рис. 2.6.:

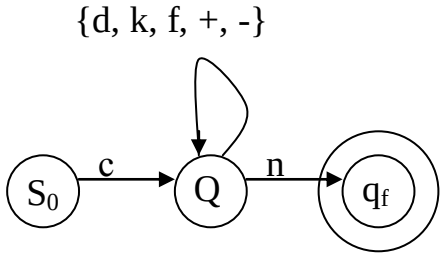


Рис. 2.6. Диаграмма переходов КА

2.6. Способы программной реализации

Способы программной реализации

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.

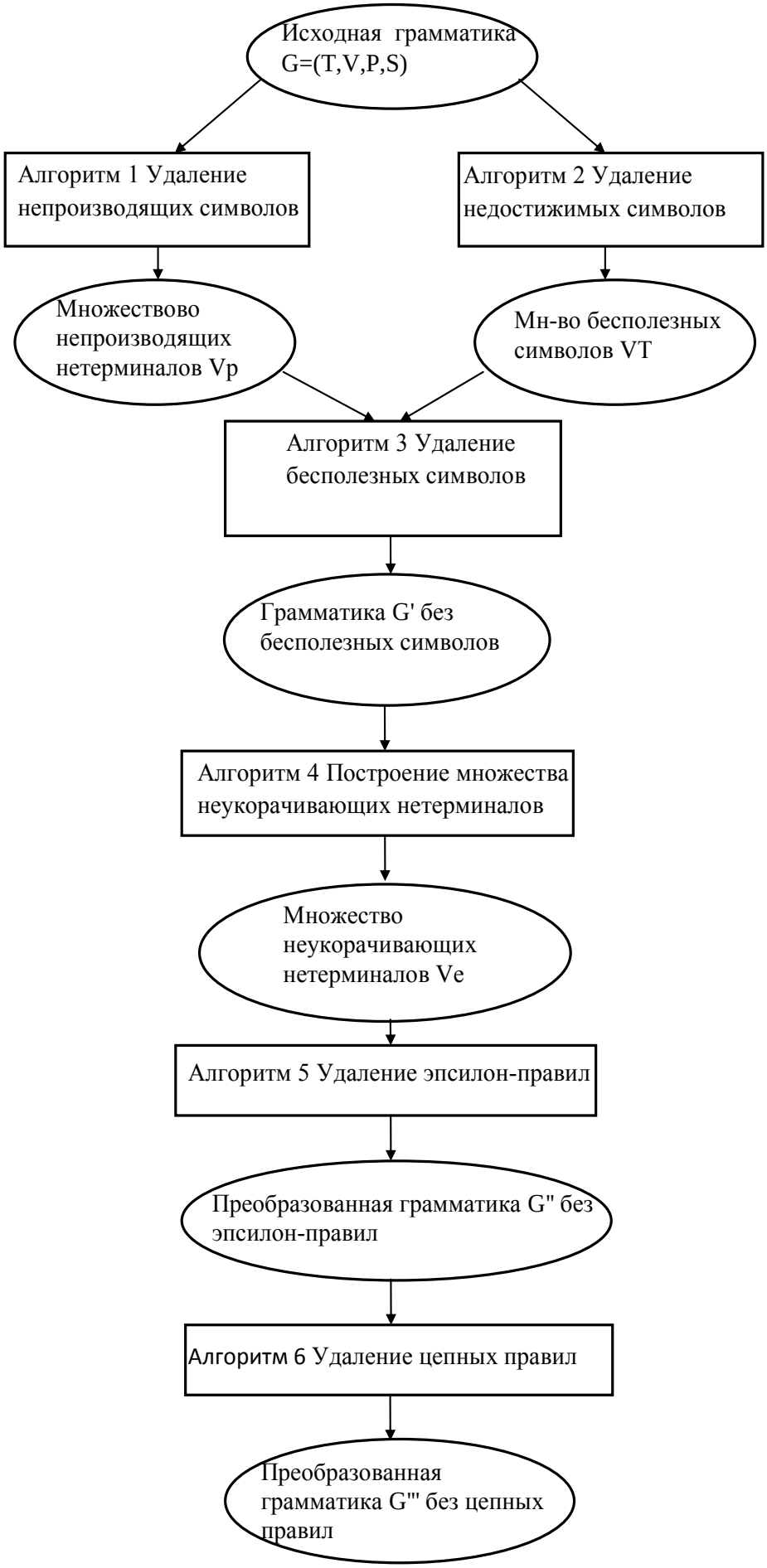


Рис.3.1 Метод преобразования грамматики к *приведенной* форме

1. Удаление непроизводящих символов

Алгоритм 3.1. 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.

Вход: КС G = (T, V, P, S)
Выход: $V_p = \{A \mid A \Rightarrow^+ \alpha, A \in V, \alpha \in T^+\}$

Алгоритм:
$V_p := \varnothing$; //множество производящих символов пустое
foreach ($A \in V$) // проходим все нетерминалы
if ($(A \rightarrow \alpha X) \in P$) //если есть производящее правило
$V_p := V_p \cup A$ //добавляем нетерминал из левой части правила в множество
end if
end foreach

Код на c#:

```
//определение множества производящих нетерминальных символов
private ArrayList producingSymb() {
      ArrayList Vp = new ArrayList();

      foreach(Prule p in this.Prules) {
            foreach(string t in this.T)
                  if (p.RightChain.Contains(t))
                        if (!Vp.Contains(p.leftNoTerm))
                        Vp.Add(p.leftNoTerm); //если еще нет - добавляем
      }
      return Vp;
}
```

2. Удаление недостижимых символов

Алгоритм 3.2. Определение *множества достижимых символов* V_r. Если нетерминал в левой части правила грамматика является достижимым, то достижимы и все символы правой части этого правила.
Вход: исходная КС G = (T, V, P, S_0), при условии $\alpha, \beta \in (V \cup T)^*$, $X \in V$
Выход: VT_r
Алгоритм:
$VT_r := \{S_0\}$
foreach ($A \in V$) //Проходим по всем нетерминалам
if ($A \rightarrow \alpha X \beta) \in P$ //Если нетерминал достижим
$VT_r := VT_r \cup X$ //Добавляем его в множество
if $\alpha \notin VT_r$ // Есть ли α в VT_r?
$VT_r := VT_r \cup \alpha$// Если нет - добавляем
end if
if $\beta \notin VT_{r\,a}$

$VT_r := VT_r \cup \beta$
end if
end if
end foreach

Код на c#:

```
//определение множества достижимых символов
private ArrayList Reachable(string StartState) {
      ArrayList Vr = new ArrayList(){ this.S0 };
//множество достижимых за 1 шаг
      ArrayList States = ReachableByOneStep(StartState);
//Нетерминальные символы из массива
      ArrayList NoTerm = NoTermReturn(States);
      Vr = Unify(Vr, NoTerm);
      foreach(string v in NoTerm) {
            Vr = Unify(Vr, ReachableByOneStep(v));
      }
      return Vr;
}
```

3. Удаление бесполезных символов

Алгоритм 3.3. Устранение *бесполезных* символов. Вначале исключить *непроизводящие* нетерминалы, а затем *недостижимые* символы.
Вход: исходная КС $G = (T, V, P, S)$, V_p, VT_r, $X \in V$, α и $\beta \in T^*$
Выход: КС $G' = (T', V', P', S)$
Алгоритм:
$P' := \varnothing$
foreach $(A \to \alpha X \beta) \in P$ // Проходим по всем правилам
if $X \in V_p$ //Если X – производящий нетерминал
$P' := P' \cup (A \to \alpha X \beta)$//Добавляем правило, из которого выводится X
end if
end foreach
$T' := T \cap VT_r$
$V' := V \cap VT_r$

Код на c#:

```
//удаление бесполезных символов
public myGrammar unUsefulDelete()
{
      ArrayList Vp = producingSymb(); //множество производящих символов
      ArrayList P1 = new ArrayList();
      ArrayList TorVp = Unify(Vp, this.T);//Объединение всех терминалов
      // и производящих символов
      foreach(Prule p in this.Prules)
      {
            ArrayList SymbInRule = SymbInRules(p); //все символы в правиле
            foreach(string s in SymbInRule)
                  if (TorVp.Contains(s) || TermReturn(p.RightChain) !=
null) {
```

```
                    if (!P1.Contains(p)) { P1.Add(p); }
                }
        }
        //построить множество достижимых символов
        ArrayList Vr = Reachable(this.S0);
        ArrayList T1 = intersection(this.T, Vr);  //пересечение множеств
        ArrayList V1 = intersection(Vr, this.V);  //пересечение множеств
        ArrayList P2 = new ArrayList();

        foreach(Prule p in P1)
        {
// Все терминалы из правила
            ArrayList SymbInRule = SymbInRules(p);
            foreach(string s in SymbInRule) {
                if (Vr.Contains(s))
                    if (!P2.Contains(p))
                        P2.Add(p);
            }
        }
        return new myGrammar(T1, V1, P2, this.S0);
}
```

4. Построение множества неукорачивающих нетерминалов

Алгоритм 3.4. Построение множества V_ε укорачивающих нетерминалов.
Вход: КС G' = (T, V, P, S).
Выход: V_ε
Алгоритм:
$V_\varepsilon := \varnothing$;
foreach A in V
  if $(A \to \varepsilon) \in$ P //Если в правой части правила ε
    $V_\varepsilon := V_\varepsilon \cup$ A//Добавляем его в множество правил
  end if
end foreach

Код на c#:

```
//построение множества укорачивающих нетерминалов
private ArrayList ShortNoTerm() {
        ArrayList Ve = new ArrayList();
        foreach(Prule p in this.Prules) {
            if (p.rightChain.Contains("")) //если правые части содержат
пустые правила
                Ve.Add(p.leftNoTerm);
        }
        return Ve;
}
```

5. Удаление эпсилон-правил

Алгоритм 3.5. Преобразование КС-грамматики с ε–правилами в эквивалентную НКС-грамматику.

Вход: КС $G' = (T, V, P, S)$, V_ε
Выход: НКС грамматика $G'' = (T, V', P', S')$
Алгоритм:
$P' := P$
foreach $(A \rightarrow \alpha_0 B_1 \alpha_1 B_j \alpha_i \ldots B_k \alpha_k) \in P'$// Проходим по всем правилам
  // Если в правой части нетерминал, из которого выводится ε-правило
  foreach $(B_k \in V_\varepsilon)$
    A_1 := delete(B_k from A) // Кладем в A1 правило A без ε-нетерминала
    $P' := P' \cup A_1$// Добавляем в P' правило A1
  end foreach
end foreach

if $(S \in V_\varepsilon)$ //если нач. символ принадлежит множеству V_ε то добавляем новое правило
  $V' = V \cup \{S'\}$
  $P' := P' \cup (S' \rightarrow S \mid \varepsilon)$
end if

Код на c#:

```
//удаление эпсилон правил
public myGrammar EpsDelete() {
      Console.WriteLine("Deleting epsylon rules");

      ArrayList Ve = ShortNoTerm();
      ArrayList P1 = new ArrayList();
      ArrayList V1 = this.V;

      foreach(Prule p in Prules) {
            if (!ContainEps(p)) {
                  P1.Add(p);
// создаем новое правило
      Prule p1 = new Prule(p.leftNoTerm, TermReturn(p.rightChain));
                  if (p.rightChain.Count != 1) {
                        P1.Add(p1);
                  }

            }
      }
      if (Ve.Contains(this.S0)) {
            V1.Add("S1");
            P1.Add(new Prule("S1", new ArrayList(){ this.S0 }));
            P1.Add(new Prule("S1", new ArrayList(){ "" }));
            return new myGrammar(this.T, V1, P1, "S1");
      }
      else
            return new myGrammar(this.T, V1, P1, this.S0);
}
```

6. Удаление цепных правил

Алгоритм 3.6. Устранение цепных правил.
Вход: КС $G'' = (T, V, P, S)$

Выход: НКС G’’’ = (T, V, P’, S)
Алгоритм:

$P_c := Ø$
$P_{nc} := Ø$
P’ := Ø

// нахождение цепных и нецепных правил
foreach (A → αBβ) ∈ P// Проходим по всем правилам
  if B ∈ V & α,β = Ø // Если A → αBβ –цепное
    Pc := Pc ∪ (A → B) // Добавляем его в мн-во цепных Pc
  else
    Pnc := Pnc ∪ (A → αBβ)//Если нет, то в добавляем в мн-во нецепных Pnc
  end if
end foreach
//замена цепных на нецепные
foreach (R→ αBβ) ∈ P
  if B→ γCδ ∈ P_c
    P’ := P’ ∪ (R→ αγCδβ)
  endif
end foreach
P’ := P’ ∪ P_{nc}

Код на c#:

```
//удаление цепных правил
        public myGrammar ChainRuleDelete() {
            Console.WriteLine("Цепные правила удалены...");

            ArrayList NoChainRules = new ArrayList();
            ArrayList ChainRules = new ArrayList();

            foreach (Prule p in this.Prules){
                if (TermReturn(p.rightChain) != null) { NoChainRules.Add(p); }
                else { ChainRules.Add(p); }
            }


            if (ChainRules.Count == 0) { Console.WriteLine("Цепных правил нет..."); }
            else
            {
                foreach (Prule chainrule in ChainRules)
                    foreach (Prule rule in NoChainRules){
                        if (rule.RightChain.Contains(chainrule.leftNoTerm))
                            for (int i = 0; i < rule.RightChain.Count; i++){
                                if (rule.RightChain[i].ToString() == chainrule.leftNoTerm)
                                    rule.RightChain[i] = chainrule.RightChain[0].ToString();
                            }
                    }
            }
            ArrayList P = new ArrayList();
            foreach (Prule rule in NoChainRules)
                if (!P.Contains(rule)) P.Add(rule);
            return new myGrammar(this.T, this.V, P, this.S0);
        }
```

Пример применения метода приведения грамматики

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

S → b | cAB

A → Ab | c
B → cB
C → Ca
F → d

Результат работы программы:

Правила:
S -> b
S -> A
S -> cAB
A -> Ab
A -> c
B -> cB
C -> Ca
F -> 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

Модификация

Данные алгоритмы можно модифицировать, объединяя похожие части. Самая простая модификация – объединение алгоритмов 1-3:

$VT_r := \{S_0\}$
$V_p := \varnothing$;
$P' := \varnothing$
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.

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

Строим рекурсивно множества V^0_p , V^1_p ,…, V^i_p ,…

1. Положить $V^0_p = \varnothing$, i=1.
2. Положить $V^i_p = V^{i-1}_p \cup \{A \mid (A \to \alpha) \in P, \alpha \in T^+\}$.
3. Если $V^i_p \neq V^{i-1}_p$, положить i = i +1 и перейти к шагу 2.
4. Положить $V_p = V^i_p$.

Поскольку $V_p \subseteq V$, то число повторений шага 2. i+1, где i – число нетерминальных символов грамматики G.

Алгоритм 3.2. Определение множества *достижимых* символов V_r. 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: $V_r = \{X \mid S \Rightarrow^* \alpha X \beta,\ X \in V,\ \alpha, \beta \in (V \cup T)^*\}$.

Строим рекурсивно множества V^0_r, V^1_r,…, V^i_r,…

1. Положить $V^0_r = \{S\}$, i=1.
2. Положить $V^i_r = V^{i-1}_r \cup \{X \mid A \rightarrow \alpha X \beta,\ X \in V,\ \alpha, \beta \in (V \cup T)^*$ и $A \in V^{i-1}_r\}$.
3. Если $V^i_r \neq V^{i-1}_r$, положить i = i +1 и перейти к шагу 2.
4. Положить $V_r = V^i_r$.

3. Символ $a \in T \cup V$ называется *бесполезным* в КС-грамматике G, если он *непроизводящий* или *недостижимый*.

Алгоритм 3.3. Устранение *бесполезных* символов. Вначале исключить *непроизводящие* нетерминалы, а затем *недостижимые* символы.

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

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

1. Построить множество V_p производящих нетерминалов грамматики G. Алгоритм 3.1.
2. Положить $G_1 = (T, V_p, P_1, S)$, где P_1 состоит из правил множества P, содержащих только символы из $T \cup V_p$ (алгоритмы 3.1.)
3. Построить множество V_r (алгоритм 3.2.), достижимых символов грамматики G_1.
4. Положить G’ = (T’, V’, P’, S), где $T' = T \cap V_r$, $V' = V_r \cap V$, P’ состоит из правил множества P_1, содержащих только символы из множества V_r.

Алгоритмы 3.1 и 3.2 следует употреблять в указанном порядке.

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

КС-грамматика называется *неукорачивающей* КС-грамматикой (НКС-грамматикой, КС-грамматикой без ε–правил) при условии, что P не содержит $S \rightarrow \varepsilon$ и S не встречается в правах частях остальных правил.

Алгоритм 3.4. Построение множества V_ε укорачивающих нетерминалов.

Вход: КС грамматика G = (T, V, P, S).

Выход: $V_\varepsilon = \{A \mid A \Rightarrow^+ \varepsilon,\ A \in V\}$.

Строим рекурсивно множества укорачивающих нетерминалов V^0_ε , V^1_ε ,…, V^i_ε ,…

1. Положить $V^0_\varepsilon = \{A \mid A \rightarrow \varepsilon,\ A \in V\}$, i=1.
2. Положить $V^i_\varepsilon = V^{i-1}_\varepsilon \cup \{ A \mid (A \rightarrow \alpha) \in P$ и $\alpha \in (V^{i-1}_\varepsilon)^+\}$.
3. Если $V^i_\varepsilon \neq V^{i-1}_\varepsilon$, положить i = i +1 и перейти к шагу 2.
4. Положить $V_\varepsilon = V^i_\varepsilon$.

Поскольку $V_\varepsilon \subseteq V$, то число повторений шага 2. i+1, где i – число нетерминальных символов грамматики G.

Алгоритм 3.5. Преобразование КС-грамматики с ε–правилами в эквивалентную НКС-грамматику.

Вход: КС грамматика G = (T, V, P, S).

Выход: НКС грамматика G’ = (T, V’, P’, S’), порождающая язык L(G’) = L(G).

1. Построить множество V_ε, укорачивающих нетерминалов (алгоритмы 3.4.).
2. Положить P’ = Ø.
3. Если $(A\rightarrow \alpha_0B_1\alpha_1B_i\alpha_i \ldots B_k\alpha_k)\in P$, где $k \geq 0$, $B_j \in V_\varepsilon$ ($1\leq j \leq k$) и ни один из символов цепочек $\alpha_i \in (V \cup T)^*$ ($0\leq i \leq k$) не содержит символов из V_ε, то включить в P’, все правила вывода вида $A\rightarrow \alpha_0X_1\alpha_1X_j\alpha_i \ldots X_k\alpha_k$, где $X_j = B_j$ или $X = \varepsilon$.

   Если все цепочки $\alpha_i = \varepsilon$, то правила $A\rightarrow \varepsilon$ не включать в P’.
4. Если $S \in V_\varepsilon$, то положить $V’ = V \cup \{S’\}$ и добавить в P’ два правила: $S’\rightarrow S$ и $S’\rightarrow \varepsilon$. В противном случае положить V’ = V и S’= S.
5. Положить G’ = (T, V’, P’, S’).

В КС грамматиках часто встречаются правила, правая часть которых состоит из одного нетерминального символа. Правило вида $A \rightarrow B$ называется цепным правилом.

Алгоритм 3.6. Устранение цепных правил.

Вход: КС грамматика G = (T, V, P, S).

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

1. Для каждого $A \in V$, построить множество $V_A = \{B \mid A \Rightarrow^+ B\}$ следующим образом:
   1.1. Положить $V^0_A = \{A\}$, i=1.
   1.2. Положить $V^i_A = V^{i-1}_A \cup \{C \mid (B\rightarrow C)\in P \text{ и } B \in V^{i-1}_A\}$.
   1.3. Если $V^i_A \neq V^{i-1}_A$, положить $V_A = V^i_A$.
2. Положить P’ = Ø.
3. Если $B\rightarrow\alpha \in P$ и не является цепным правилом, то положить P’= P’ $\cup \{A\rightarrow\alpha\}$ для всех таких A, что $B \in V_A$.
4. Положить G’ = (T, V, P’, S).

Нетерминал КС-грамматики называется *рекурсивным*, если $A \Rightarrow^+ \alpha A\beta$, для некоторых α и β. Если α = ε, то A называется *леворекурсивным*, если β = ε, 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.

Алгоритм 3.7.

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

Выход: Эквивалентная КС грамматика G’.

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

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

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

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

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

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 $A_i \to \beta_1\alpha |\ldots| \beta_m\alpha$, где $A_j \to \beta_1 |\ldots| \beta_m$ – 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.

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 $A \to \varepsilon$, и каждое правло из P отличное от $S \to \varepsilon$, имеет вид $A \to a\alpha$, 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. $A \to BC$, где $A, B, C \in V$;
2. $A \to a$, где $a \in T$;
3. $S \to \varepsilon$, если $\varepsilon \in L(G)$, причем S не встречается в правых частях правил.

3.3. Пример: приведенная форма КС-грамматики

Постановка задачи. Привести КС-грамматику к *приведенной* форме, исключить левую рекурсию.

А).Устранить из грамматики G бесполезные символы. Применить алгоритм 3.3. к грамматике G = (T, V, P, S), где $V = \{A,B,C,S\}$, $T = \{a,b,c\}$, $P = \{S \to aC, S \to A, A \to cAB, B \to b, C \to a\}$.

Шаг 1. множество $V_p = \{B, C, S\}$ A – *непроизводящий символ*

Шаг 2. $G_1 = (\{B, C, S\}, \{a,b,c\}, P, S)$, где $P = \{S \to aC, B \to b, C \to a\}$,

Шаг 3. $V_r = \{C, S\}$ B – *недостижим, так как* A- *непроизводящий символ*

Шаг 4. $G' = (\{C, S\}, \{a\}, P, S)$, где $P = \{S \to aC, C \to a\}$, $L(G') = \{aa\}$

В). Устранить из грамматика G ε–правила, применить алгоритм 3.5. Преобразовать грамматику G = (T, V, P, S), где $V = \{A,S\}$, $T = \{b,c\}$, $P = \{S \to cA, S \to \varepsilon, A \to cA, A \to bA, A \to \varepsilon\}$, в эквивалентную НКС-грамматику.

Шаг 1. Применяя алгоритм 3.4., 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.

Шаг 3. Алгоритм 3.5. 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 4. $S \in V_\varepsilon$, то в V’ добавляем новый нетерминал S’, а в P’ два правила S’→ S и S’→ ε.

Шаг 5. G’ = ({b,c}, { S’, S, A} , { S’→ S, S’→ ε, S→cA, S→c, A→cA, A→c, A→bA, A→b}, S’).

C). Устранить из КС грамматики G цепные правила, применить алгоритм 3.6. G = ({S,F,L}, {v,+,*,(,)}, P, S), где P состоит из правил

S → S + F | F, F → F * L | L, L → v | (S)

Шаг 1. Применяя алгоритм 3.6., получаем V_S = {S, F, L}, V_F = {F, L}, V_L = {L}.

Шаг 2. Положить P’ = Ø.

Шаг 3. Выберем первый нетерминал S из множества V_S. Множество правил, левая часть которых – нетерминальнй символ S, правые части - это правые части нецепных правил исходной грамматики, в левой части которых находятся символы из множества V_S , получаем: {S→ S + F | F*L | (S) | v }. Включаем эти правила в P’={S→ S + F | F*L | (S) | v }. Таким же образом рассматриваем символы из V_F , V_L ,

Шаг 4. В результате G’ = (T, V, P’, S), где P’={S→ S + F | F*L | (S) | v , F → F * L | (S) | v, L → v | (S)}

D). Исключить левую рекурсию из КС – грамматики G.

1.Устранить левую рекурсию в заданной КС-грамматике G, порождающей скобочные арифметические выражения.

G = ({v,+,*,(,)}, {S,F,L}, P, S), где P состоит из правил

S → S + F | F

F → F * L | L

L → v | (S)

Применить алгоритм 3.7. к грамматике G. Указаны шаги алгоритма 3.7.

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 → S + F| F, α = + F, β = F. Заменим S - правила правилами : S → F| FS’, и добавим в грамматику правило для нового нетерминала S’→ + F | + FS’.

Шаг 3. i = 2 и j = 1.

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

Шаг 5. Так как j = i – 1, то переходим к **шагу 2**.

Шаг 2. Для i = 2 преобразуем правила: F → F * L | L , α = * L , β = L. Заменим F - правила правилами: F→ L| LF’, и добавим в грамматику правило для нового нетерминала F’→ * L| LF’ .

Шаг 3. i = 3, j = 1.

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

Шаг 5. Так как $j \neq i-1$, то $j = j + 1$ переходим к **шагу 4.**

Шаг 4. Для $i = 3$, $j = 2$ правила вида $A_i \rightarrow A_j\alpha$ отсутствуют.
Шаг 5. Так как $j = i - 1$, то переходим к **шагу 2.**

Шаг 2. Для $i = 3$ правила вида $A_i \rightarrow A_i\alpha$ отсутствуют.
Шаг 3. $i = n = 3$, преобразование завершено.
Получили правила новой грамматики G':
S→ F| FS', S'→ + F | + FS', F→L | LF', F'→ * L| LF', L → (S + F) | v |(S).

F) Определить в какой форме (Грейбах, Хомского) находится КС-грамматика G'.

E). G' – *приведенная* КС-грамматика.

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.

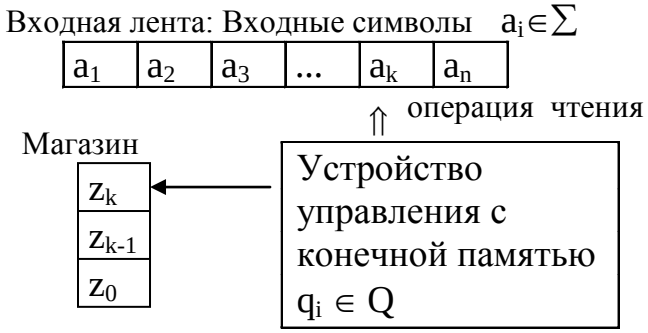


Рис. 1.10. Модель МП - автомата

Определение 9. МП автомат – это семерка объектов

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

Q – конечное множество состояний устройства управления;

Σ - конечный алфавит входных символов;

Γ - конечный алфавит магазинных символов;

δ - функция переходов, отображает множества $Q\times(\Sigma \cup \{\varepsilon\})\times\Gamma$ 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 $(q, \omega, z) \in Q\times\Sigma^*\times\Gamma^*$, 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 = ε, 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 ≠ ε, 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 = ε, 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 $\omega \in \sum^*$, 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 $\omega \in \sum^*$ допускается МП-автоматом МП =$(Q, \sum, \Gamma, \delta, q_0, z_0, F)$, 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:

$L(МП) = \{\omega \mid \omega \in \sum^*$ 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 = $(\{q_0, q_1, q_2, q_f\}, \{a,b\}, \{ z_0,a\}, \delta, q_0, z_0, \{ q_f \})$

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

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

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

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

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

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}, $\sum$, $\sum \cup$ 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 $a \in \sum$.

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

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. $\delta(q_0, \varepsilon, S) = \{(q, S + F), (q, F)\}$;
2. $\delta(q, \varepsilon, F) = \{(q, F * L), (q, L)\}$;
3. $\delta(q, \varepsilon, L) = \{(q, v), (q, (S))\}$;
4. $\delta(q, a, a) = \{(q, \varepsilon)\}$ для всех $a \in \sum = \{v,+,*,(,)\}$.

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

$(q_0, v * (v+ v), S) \vdash_1 (q, v * (v+ v), F) \vdash (q, v * (v+ v), F*L) \vdash (q, v * (v+ v), L*L) \vdash (q, v * (v+ v), v*L) \vdash (q, * (v+v), *L) \vdash (q, (v+v), L) \vdash (q, (v+v), (S)) \vdash (q, v+v), S)) \vdash (q, v+v), S+F)) \vdash (q, v+v), F+F)) \vdash (q, v+v), L+F)) \vdash (q, v+v), v+F)) \vdash (q, +v), +F)) \vdash (q, v), F)) \vdash (q, v), L)) \vdash (q, v), v)) \vdash (q,),)) \vdash (q, \varepsilon, \varepsilon)$.

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, $\sum$, Γ, δ, q_0, z_0, F), где верхним элементом магазина является самый правый символ цепочки, δ-функция переходов, которая задает

отображение множества $Q\times(\sum \cup \{\varepsilon\})\times\Gamma$ в множество конечных подмножеств множества $Q\times\Gamma^*$, 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 $\perp$ - конец цепочки. Пусть РМП = ({q,r}, $\sum$, $\sum \cup V \cup \perp$, δ, q, $\perp$, {r}), где δ определяется следующим образом:

1. $\delta(q, a, \varepsilon) = \{(q, a)\}$ для всех $a \in \sum$ (символы с входной ленты на этих тактах переносятся в магазин).
2. Если $A \rightarrow \alpha$ - правило вывода грамматики G, то $\delta(q, \varepsilon, \alpha) = (q, A)$.
3. $\delta(q, \varepsilon, \perp S) = \{(r, \varepsilon)\}$

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 $(Q, \sum, \Gamma, \delta, q_0, z_0, F)$, используем алгоритм 3.9, получим РМП = ({q,r}, {v,+,*,(,)}, {v,+,*,(,), S,F,L,$\perp$}, δ, q, $\perp$, {r}), где функция переходов δ определена следующим образом:

1. $\delta(q, a, \varepsilon) = \{(q, a\}$ для всех $a \in \sum$={v,+,*,(,)};
2. $\delta(q, \varepsilon, S+F) = \{(q, S)\}$;
3. $\delta(q, \varepsilon, F) = \{(q, S)\}$;
4. $\delta(q, \varepsilon, F * L) = \{(q, F)\}$;
5. $\delta(q, \varepsilon, L) = \{(q, F)\}$;
6. $\delta(q, \varepsilon, v) = \{(q, L)\}$;
7. $\delta(q, \varepsilon, (S)) = \{(q, L)\}$;
8. $\delta(q, \varepsilon, \perp S) = \{(r, \varepsilon)\}$;

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

$(q, v+v*v, \perp) \vdash^1 (q,+v*v, \perp v)$
$\vdash^6 (q, +v*v, \perp L)$
$\vdash^5 (q, +v*v, \perp F)$
$\vdash^3 (q, +v*v, \perp S)$
$\vdash^1 (q, v*v, \perp S+)$
$\vdash^1 (q, *v, \perp S+v)$
$\vdash^6 (q, *v, \perp S+L)$
$\vdash^5 (q, *v, \perp S+ F)$
$\vdash^1 (q, v, \perp S+ F*)$
$\vdash^1 (q, \varepsilon, \perp S+ F*v)$
$\vdash^6 (q, \varepsilon, \perp S+ F*L)$
$\vdash^4 (q, \varepsilon, \perp S+F)$
$\vdash^2 (q, \varepsilon, \perp S)$
$\vdash^8 (r, \varepsilon, \varepsilon)$

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 $q \in Q$ и $z \in Г$ выполняется одно из условий:

1. δ(q, a, z) содержит не более одного элемента для каждого $a \in \sum$ и δ(q, ε, z) = Ø;
2. δ(q, a, z) = Ø для всех $a \in \sum$, 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:

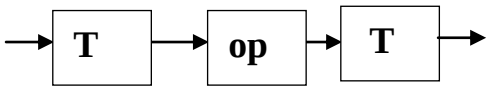
E ⇒ T op T ⇒ T op (E) ⇒ F op (T op T) ⇒ a op (T op T) ⇒ a + (T op T) ⇒ a + (F op T) ⇒ a + (b op T) ⇒ a + (b op F) ⇒ a + (b op c) ⇒ a + (b - c)

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

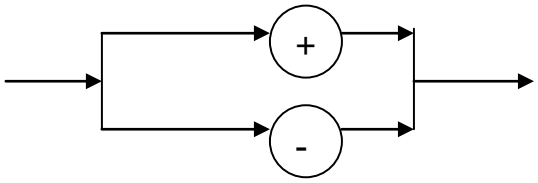
p_1: E := { T op T}, p_2: op := {+,-}, p_3: T := {F , (E)}, p_4: F :={a, b, c}

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 А можно пройти, не читая вообще никаких входных символов, то такая "нулевая ветвь" должна помечаться всеми символами, которые могут следовать за А (это необходимо для принятия решения о переходе на эту ветвь).

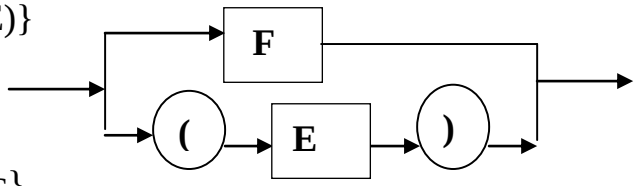
p_1: E:= {T op T}



p_2: op:= {+,-}



p_3: T := {F , (E)}



p_4: F :={a, b, c}

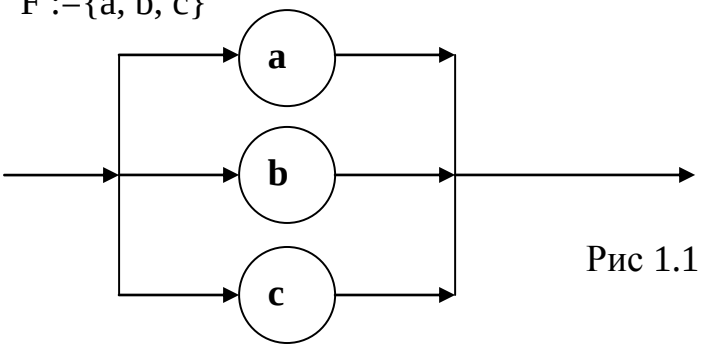
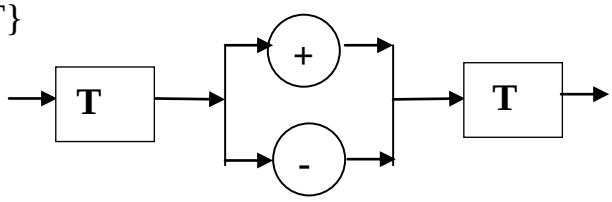


Рис 1.11. Диаграммы Вирта.

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:

p_1: E:= {T {+,-} T}



p_4: T := {{a, b, c}, (E)}

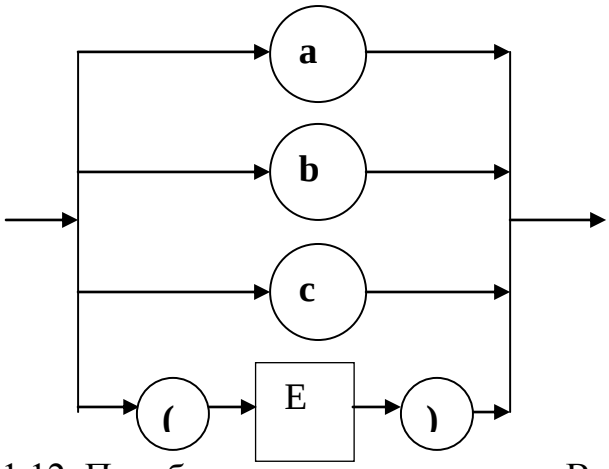


Рис 1.12. Преобразованные диаграммы Вирта.

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:

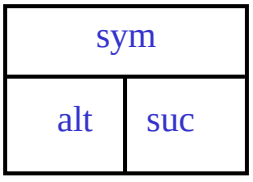


Рис. 1. 14. Структура узла Node

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.

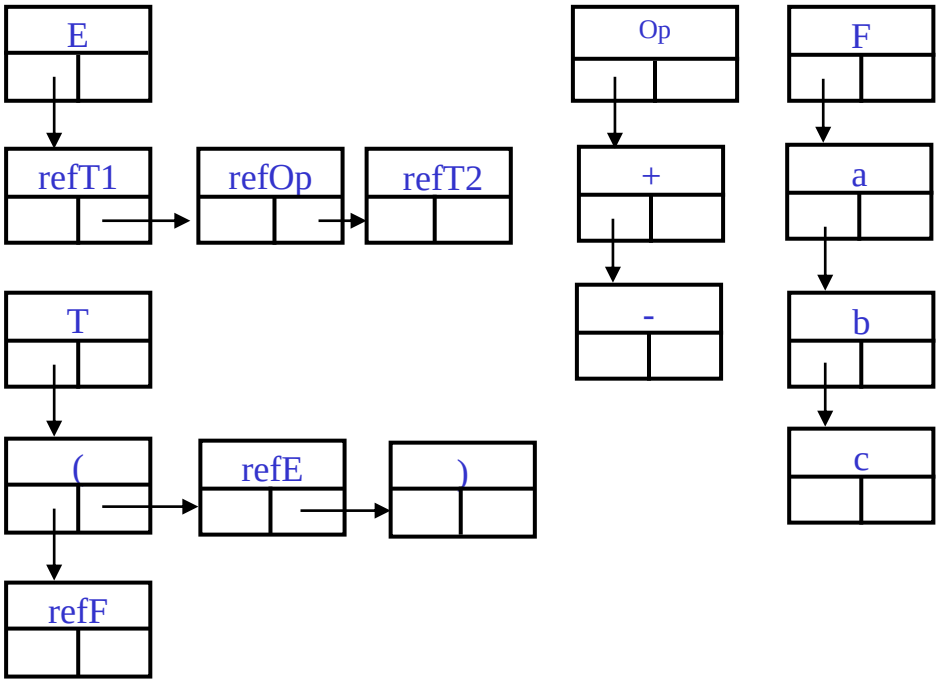


Рис. 1.15. Структура данных для диаграмм п.5

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:

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

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

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

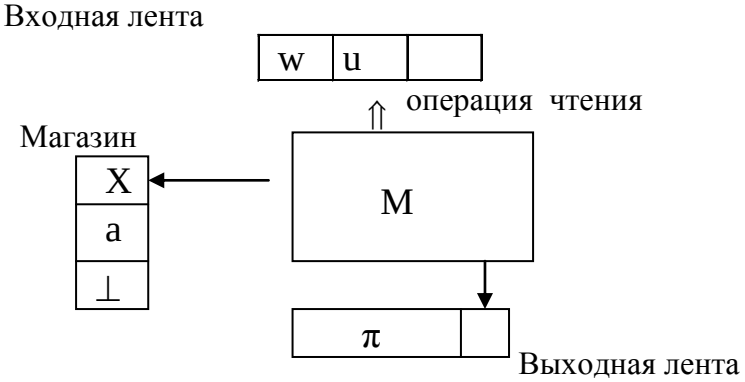


Рис. 1.18. LL(k)-анализатор

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) = M(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', +) = M(+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) =M(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) = M(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) ⊢ (i*i, TE’⊥, 14862) ⊢ (i*i, PT’E’⊥, 148624) ⊢ (i*i, iT’E’⊥, 1486248) ⊢ (*i, T’E’⊥, 1486248) ⊢	M(T’, +) = (ε, 6) M(E’,+) = (+ TE’,2) M(+,+) = ВЫБРОС M(T, i) = (PT’, 4) M(P, i) = (i, 8) M(i,i) = ВЫБРОС M(T’, *) = (*PT’, 5)

(*i, *PT’E’⊥, 14862485) ⊢ (i, PT’E’⊥, 14862485) ⊢ (i, iT’E’⊥, 148624858) ⊢ (ε, T’E’⊥, 148624858) ⊢ (ε, E’⊥, 1486248586) ⊢ (ε, ⊥, 14862485863)	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.

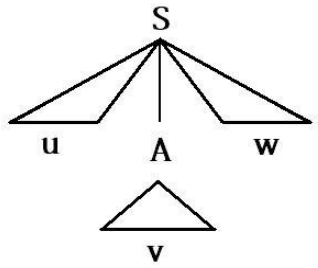
3.7. LR(k)-грамматики

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

Грамматики, для которых можно построить детерминированный восходящий анализатор, получили название LR(k)-*грамматик* (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).

Наиболее наглядно LR(k)-*грамматику* можно определить в терминах деревьев вывода.

Определение 14. КС - грамматика G = (T, V, P, S) является LR(k)-*грамматикой*, если просмотрев только часть кроны дерева вывода в этой грамматике, расположенной слева от данной внутренней вершины, часть кроны, выведенную из нее, и следующие k символов входной цепочки, можно установить правило вывода, которое было применено к этой вершине при порождении входной цепочки.



Дерево вывода цепочки uvw.

В определении LR(k)-*грамматики* используется

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:

$FIRST_k(\gamma) = \{x \mid \gamma =>^*_l xw$ и $|x| = k$ или $\gamma =>^*_l x$ 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 $k \geq 0$, если из существования двух правых выводов для *пополненной* грамматики G' = (T, V', P', S') полученной из G:

$S' \Rightarrow^*_r \alpha Aw \Rightarrow_r \alpha\beta w$,

$S' \Rightarrow^*_r \gamma Bx \Rightarrow_r \alpha\beta y$,

для которых $FIRST_k(w) = FIRST_k(y)$ следует, что $\alpha Ay = \gamma Bx$.

То есть, если αβw и αβy – правовыводимы 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**: 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.

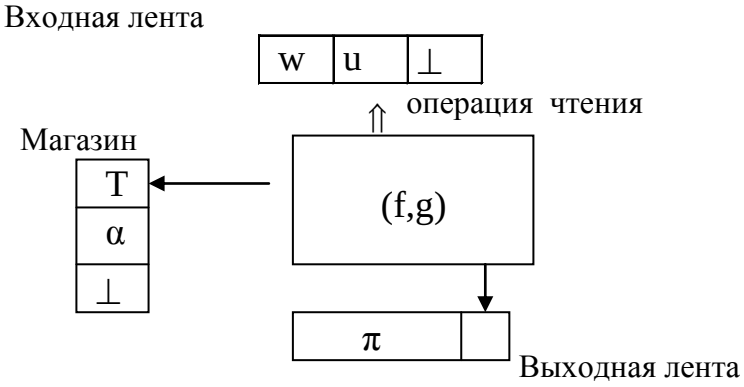


Рис. 1.18. LR(k)-анализатор общий вид

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

Рассмотрим алгоритм "перенос-свертка", выполняемый детерминированным восходящим синтаксическим анализатором.

Алгоритм состоит в переносе входных символов в магазин до тех пор, пока в его верхней части не встретится основа.

В этот момент производится свертка, в результате которой основа заменяется левой частью основывающего правила. Этот процесс продолжается до тех пор, пока не будет прочитана вся входная цепочка, а в магазине не останется начальный символ грамматики или алгоритм не выдаст сообщение об ошибке.

Пусть в КС-грамматике G имеется правовыводимая цепочка $S =>^*_r \alpha x$, такая, что цепочка α заканчивается нетерминальным символом (или это пустая цепочка).

Назовем цепочку α открытой частью цепочки αx, а x – замкнутой частью цепочки. Граница между открытой и замкнутой частью цепочки называется рубежом.

Алгоритм "перенос-свертка" можно рассматривать как программу работы расширенного детерминированного МП-преобразователя.

Рассмотрим правый вывод $S =>_r \alpha_0 =>_r \alpha_1 =>_r \ldots =>_r \alpha_m = w$. Допустим, что $\alpha_{i-1} = \gamma Bz$ и на i-том шаге вывода $\alpha_{i-1} =>_r \alpha_i$ было применено правило B → βy

грамматики. Пусть $\gamma B = \alpha A$ и $yz = x$. Тогда цепочка $\alpha_i = \gamma\beta yz = \alpha Ax$. При этом преобразователь моделирует обращение вывода $\alpha_{i-1} <= \alpha_i$.

В начале моделирования i-го шага состояние ДМП-преобразователя определяется цепочкой α_i: в магазине находится αA, а на входной ленте – x. Выполняя разбор, преобразователь будет переносить символы из цепочки x в магазин, пока не определит правый конец основы (возможно, ни одного переноса не потребуется). К этому моменту в магазин будет перенесна цепочка y и в нем будет находиться цепочка $\alpha Ay = \gamma By$.

Преобразователь выполнит свертку, используя правило $B \rightarrow \beta y$, в магазине появится цепочка γB – открытая часть правовыводимой цепочки α_{i-1}, а на входной ленте останется цепочка z – замкнутая часть α_{i-1}.

Моделирование шага вывода завершено.

Два способа построения LR(k) анализаторов

Два способа:

1. Определения активных префиксов - использование расширенного магазинного алфавита (алгорит перенос-свертка)
2. На основе грамматического вхождения.

Использование расширенного магазинного алфавита: алгоритм перенос-свертка

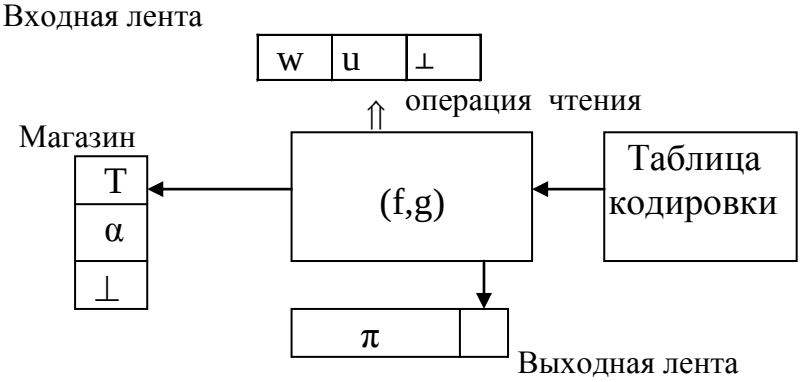


Рис. 1.18. LR(k)-анализатор

Шаг 1. Использование расширенного магазинного алфавита. Анализируя верхний символ магазина определяется наличие в нем основы и правила грамматики для свертки.

1. S→ (AS) 3. A→ (SaA)
2. S→ (b) 4. A→ (a)

Каждый магазинный символ алгоритма содержит сведения о символе грамматики «кодируемой цепочки» (состояние алгоритма). Пример кодирования магазинных символов для рассмотренной ранее грамматики приведен в таблице 3.12.1. Например, магазинный символ «$)_3$», 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 «(SaA)», то есть *основа* - правая часть правила, 3.

Таблица 3.12.1. Кодирование магазинных символов.

Символ грамматики	Магазинный символ	Кодируемая цепочка
S	S_1 S_2 S_3	(AS (S ⊥S
A	A_1 A_2	(A (SaA
a	a_1 a_2	(Sa (a
b	b_1	(b
(	$(_1$	(
)	$)_1$ $)_2$ $)_3$ $)_4$	(AS) (b) (SaA) (a)
_	⊥	⊥

Магазинный алфавит построен таким образом, что для каждого магазинного символа (за исключением S_3 и ⊥), кодируемая им цепочка является *префиксом правой части* некоторого правила грамматики. И наоборот, каждый непустой префикс правой части правила кодируется некоторым магазинным символом.

Например, правая часть правила грамматики:

(1) S→ (AS)

имеет четыре непустых префикса: «(», «(A», «(AS», «(AS)», которые кодируются четырьмя магазинными символами: «$(_1$», «A_1», «S_1», «$)_1$» соответственно.

Шаг 2. 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.

Например, если в магазине находится цепочка $⊥(_1(_1S_2a_1$, то алгоритм может втолкнуть в магазин только символ A_2, так как после этого в магазине будет находиться цепочка $⊥(_1(_1S_2a_1A_2$, суффиксом которой является цепочка, кодируемая символом A_2, а именно (SaA.

Управление алгоритмом осуществляется при помощи двух функций, приведенных в таблице 8.2, следующим образом:

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

Таблица 8.2. Функции управления алгоритмом.

		функция действий f(u)				функции переходов g(X)					
	(	a	b	)	⊥	S	A	a	b	(	)
S_1	П	П	П	П						$(_1$	$)_1$
S_2	П	П	П	П				a_1		$(_1$	
S_3	П	П	П	П	ДОП					$(_1$	
A_1	П	П	П	П		S_1				$(_1$	$)_3$
A_2	П	П	П	П						$(_1$	
a_1	П	П	П	П			A_2		S_3	$(_1$	
a_2	П	П	П	П						$(_1$	$)_4$
b_1	П	П	П	П						$(_1$	$)_2$
$(_1$	П	П	П	П		S_2	A_1	a_2	b_1	$(_1$	
$)_1$	C(1)	C(1)	C(1)	C(1)	C(1)						
$)_2$	C(2)	C(2)	C(2)	C(2)	C(2)						
$)_3$	C(3)	C(3)	C(3)	C(3)	C(3)						
$)_4$	C(4)	C(4)	C(4)	C(4)	C(4)						
⊥	П	П	П	П		S_3				$(_1$	

Пример 3.12.1

Рассмотрим работу алгоритма при разборе цепочки (((b)a(a))((a)(b))). Эта цепочка принадлежит языку, определяемому рассматриваемой грамматикой, так как ее правый вывод:

$S \Rightarrow_{(1)} (AS) \Rightarrow_{(1)} (A(AS)) \Rightarrow_{(2)} (A(A(b))) \Rightarrow_{(4)} (A((a)(b))) \Rightarrow_{(3)} ((SaA)((a)(b)))$
$\Rightarrow_{(4)} ((Sa(a))((a)(b))) \Rightarrow_{(2)} (((b)a(a))((a)(b)))$

Шаг 1. Начальная конфигурация алгоритма (⊥, (((b)a(a))((a)(b))), ε).

Функция действий f(u) определяется верхним символом магазина и входным символом. Для параметров начальной конфигурации (дно магазина «⊥» и первый входной символ «(») функция действия равна ПЕРЕНОС, f(u)=П.

Определяем магазинный символ, который должен быть помещен в магазин. Так как значение функции переносов для тех же параметров «⊥» и «(» равно магазинному символу «$(_1$», то алгоритм переходит в конфигурацию:

$(\perp(_1, ((b)a(a))((a)(b))), \varepsilon)$.

Шаг 2. Выполняя аналогичные действия алгоритм проходит следующие конфигурации:

$(\perp,(((b)a(a))((a)(b))), \varepsilon) \vdash_s (\perp(_1,((b)a(a))((a)(b))), \varepsilon)$

$\vdash_s (\perp(_1(_1,(b)a(a))((a)(b))), \varepsilon)$
$\vdash_s (\perp(_1(_1(_1,b)a(a))((a)(b))), \varepsilon)$
$\vdash_s (\perp(_1(_1(_1b_1,)a(a))((a)(b))), \varepsilon)$
$\vdash_s (\perp(_1(_1(_1b_1)_2,a(a))((a)(b))), \varepsilon)$

Шаг 3. Алгоритм находится в конфигурации $(\perp(_1(_1(_1b_1)_2,a(a))((a)(b))), \varepsilon)$. Значение функции действий для аргументов «$)_2$» и символа «a» равно СВЕРТКА(2). Для выполнения свертки по второму правилу грамматики мы должны вычеркнуть из магазина 3 верхних символа и втолкнуть в него некоторый магазинный символ.

Для нахождения значения вталкиваемого символа нужно определить значение функции переходов для следующих аргументов: левой части правила, указанного в значении функции действий и входного символа.

В данном случае это левая часть второго правила грамматики S и входного символа «a», следовательно, получаем значение S_2, которое помещаем в верхушку магазина.

Шаг 4. Продолжая работу, алгоритм пройдет следующие конфигурации:

$(\perp(_1(_1(_1b_1)_2,a(a))((a)(b)))\perp, \varepsilon)$
$\vdash_{r(2)} (\perp(_1(_1S_2,a(a))((a)(b)))\perp, 2)$
$\vdash_s (\perp(_1(_1S_2a_1,(a))((a)(b)))\perp, 2)$
$\vdash_s (\perp(_1(_1S_2a_1(_1,a))((a)(b)))\perp, 2)$
$\vdash_s (\perp(_1(_1S_2a_1(_1a_2,))((a)(b)))\perp, 2)$
$\vdash_s (\perp(_1(_1S_2a_1(_1a_2)_4,)((a)(b)))\perp, 2)$
$\vdash_{r(4)} (\perp(_1(_1S_2a_1A_2,)((a)(b)))\perp, 24)$
$\vdash_s (\perp(_1(_1S_2a_1A_2)_3,((a)(b)))\perp, 24)$
$\vdash_{r(3)} (\perp(_1A_1,((a)(b)))\perp, 243)$
$\vdash_s (\perp(_1A_1(_1,(a)(b)))\perp, 243)$
$\vdash_s (\perp(_1A_1(_1(_1,a)(b)))\perp, 243)$
$\vdash_s (\perp(_1A_1(_1(_1a_2,)(b)))\perp, 243)$
$\vdash_s (\perp(_1A_1(_1(_1a_2)_4,(b)))\perp, 243)$
$\vdash_{r(4)} (\perp(_1A_1(_1A_1,(b)))\perp, 2434)$
$\vdash_s (\perp(_1A_1(_1A_1(_1,b)))\perp, 2434)$
$\vdash_s (\perp(_1A_1(_1A_1(_1b_1,)))\perp, 2434)$
$\vdash_s (\perp(_1A_1(_1A_1(_1b_1)_2,))\perp, 2434)$
$\vdash_{r(2)} (\perp(_1A_1(_1A_1S_1,))\perp, 24342)$
$\vdash_s (\perp(_1A_1(_1A_1S_1)_1,)\perp, 24342)$
$\vdash_{r(1)} (\perp(_1A_1S_1,)\perp, 243421)$
$\vdash_s (\perp(_1A_1S_1)_1, \perp, 243421)$
$\vdash_{r(1)} (\perp S_3, \perp, 2434211)$
$\vdash_s$ ДОПУСК.

Шаг 5. Алгоритм успешно завершил работу и выдал *правый разбор* входной цепочки (((b)a(a))((a)(b))), равный 2434211.

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 $T'V_p\{\perp\}$, 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 $T'V_p\{\perp\}$, 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.

2. Грамматическое вхождение

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

Пример: грамматическое вхождение $A_{5,2}$ = ?

(1): $S \rightarrow aAb$

(2): $S \rightarrow c$

(3): $A \rightarrow bS$

(4): $A \rightarrow bB$

(5): $B \rightarrow aA$

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.

Пример.

Так как LR(0) - ситуация - это просто правило грамматики с точкой в некоторой позиции правой части, то для правила P → (E) можно получить 4 ситуации:

[P → • (E)]
[P → (•E)]
[P → (E•)]
[P → (E)•]

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

Из определения LR(k)-ситуации следует, что 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 расширенную грамматику G'_0, имеющую следующие правила:

0. *E'* → *E*
1. *E* → *E* +*T*
2. *E* → *T*
3. *T* → *T* **P*
4. *T* → *P*
5. *P* → *i*
6. *P* → *(E)*

Пусть вначале множество ситуации CLOSURE(I) содержит одну ситуацию $E' \rightarrow \bullet E$, т.е. CLOSURE(I) = { $E' \rightarrow \bullet E$ }

Ситуация E'→ • E содержит точку перед символом E, поэтому по второму правилу в CLOUSURE(*I*) необходимо включить *E*-правила с точкой слева:

$\boldsymbol{T \rightarrow \cdot T^{*} P,\ T \rightarrow \cdot P,\ P \rightarrow \cdot i,\ P \rightarrow \cdot (E)}$

Окончательно получим:

CLOSURE(*I*) =$\boldsymbol{\{E' \rightarrow \cdot E,\ E \rightarrow \cdot E+T,\ E \rightarrow \cdot T,\ T \rightarrow T^{*}P,\ T \rightarrow \cdot P,\ P \rightarrow \cdot i,\ P \rightarrow \cdot (E)\}}$**.**

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

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

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

Пусть для грамматики $\boldsymbol{G'_0}$ множество $I = \{[\boldsymbol{E' \rightarrow E \cdot}],[\boldsymbol{E \rightarrow E \cdot + T}]\}$.

Для вычисления значения GOTO(*I*,+) необходимо рассмотреть ситуации, в которых сразу за точкой идет символ (+). Это ситуация $[\boldsymbol{E \rightarrow E \cdot + T}]$. Перемещая точку за символ (+), построим множество ситуаций $\{[\boldsymbol{E \rightarrow E + \cdot T}]\}$ и замыкание этого множества:

$\{\boldsymbol{T \rightarrow \cdot T^{*} P,\ T \rightarrow \cdot P,\ P \rightarrow \cdot i,\ P \rightarrow \cdot (E)}\}$

Тогда GOTO(*I*,+) = $\{[\boldsymbol{E \rightarrow E \cdot + T}],\ [\boldsymbol{T \rightarrow \cdot T^{*} P}],\ [\boldsymbol{T \rightarrow \cdot P}],\ [\boldsymbol{P \rightarrow \cdot i}],\ [\boldsymbol{P \rightarrow \cdot (E)}]\}$

В [2] и [3] приведены алгоритмы. Позволяющие построить каноническую систему множеств *LR(0)* –ситуаций φ. 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($[S' \rightarrow \cdot S]$), которое в начале «не отмечено».

3. Если множество ситуаций *A*,входящее в систему φ, «не отмечено», то:

- отметить множество *A*;
- вычислить для каждого символа $X \in (\mathrm{N} \cup \Sigma)$ значение A'= GOTO(*A*,*X*);
- если множество $A' \neq \emptyset$ и еще не включено в φ, то включить его в систему множеств как φ «неотмеченное» множество.

4. Повторять шаг 3, пока все множества ситуаций системы φ не будут отмечены.

Пример 8.5.

Определим каноническую систему множеств LR(0) – ситуаций для пополненной грамматики G_0 с правилами:

(0) $E' \rightarrow E$

(1) E → E + T
(2) E → T
(3) T → T * P
(4) T → P
(5) P → *i*
(6) P → (E)

В начале построения система множеств $\varphi = \emptyset$.
Определив множество A_0 = CLOSURE({|E′→ • E|}) = {E′ → • E, E → • E + T, E → • T, T → • T * P, T → • P, P → • *i*, P → • (E)}, включим его в систему φ в качестве «неотмеченного» множества:

A_0	E′ → • E E → • E + T E → • T T → • T * P T → • P P → • (E) P → • *i*

Теперь мы должны отметить множество A_0 и определить множества GOTO(A_0, X) для всех символов грамматики G_0:

A_1 = GOTO(A_0, E) = GOTO({E′ → • E, E → • E + T}, E) =
CLOSURE({E′ → E•, E → E • + T}) = {E′ → E •, E → E • + T}.

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

A_0	E′ → • E E → • E + T E → • T T → • T * P T → • P P → • (E) P → • *i*

A_1	E → E • E → E • + T

Продолжая строить GOTO(A_0, X), получаем:

A_2 = GOTO(A_0, T) = GOTO({E → • T, T → • T * P}, T) =
CLOSURE({E → T •, T → T • * P}) = {E → T •, T → T • * P}.

Включаем A_2 в φ.

A_3 = GOTO(A_0, P) = GOTO({T → • P}, P) = {T → P •}.

Включаем A_3 в φ.

A_4 = GOTO(A_0, ( ) = GOTO({P → • (E)}, ( ) = {P → (• E), E → • E + T,
E → • T, T → • T * P, T → • P, P → • *i*, P → • (E),}.

Включаем A_5 в φ.

Остальные множества GOTO(A_0, X), где X ϵ {), +, *}, пусты, поэтому система φ принимает вид:

A_0	$E' \to \bullet E$ $E \to \bullet E + T$ $E \to \bullet T$ $T \to \bullet T * P$ $T \to \bullet P$ $P \to \bullet (E)$ $P \to \bullet i$
A_1	$E' \to E \bullet$ $E \to E \bullet + T$
A_2	$E \to T \bullet$ $T \to T \bullet * P$
A_3	$T \to P \bullet$
A_4	$P \to (\bullet E)$ $E \to \bullet E + T$ $E \to \bullet T$ $T \to \bullet T * P$ $T \to \bullet P$ $P \to \bullet (E)$ $P \to \bullet i$
A_5	$P \to i \bullet$

Продолжая выполнять шаг 3 построения, «отмечаем» множество A_1 и строим множества $A_3 = GOTO(A_1, X)$.

Кроме множества:

$A_6 = GOTO(A_1, +) = GOTO(\{E \to E \bullet + T\}, +) =$

$= CLOSURE (\{ E \to E + \bullet T\}) =$

$= \{E \to E + \bullet T, T \to \bullet T * P, T \to \bullet P, P \to \bullet i, P \to \bullet (E)\},$

Остальные множества пусты. Включаем A_6 в φ и получаем:

A_0	$E' \to \bullet E$ $E \to \bullet E + T$ $E \to \bullet T$ $T \to \bullet T * P$ $T \to \bullet P$ $P \to \bullet (E)$ $P \to \bullet i$
A_1	$E' \to E \bullet$ $E \to E \bullet + T$

A_2	$E \rightarrow T \bullet$ $T \rightarrow T \bullet * P$
A_3	$T \rightarrow P \bullet$
A_4	$P \rightarrow (\bullet E)$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * P$ $T \rightarrow \bullet P$ $P \rightarrow \bullet (E)$ $P \rightarrow \bullet i$
A_5	$P \rightarrow i \bullet$
A_6	$E \rightarrow E + \bullet T$ $T \rightarrow \bullet T * P$ $T \rightarrow \bullet P$ $P \rightarrow \bullet (E)$ $P \rightarrow \bullet i$

Продолжая выполнять шаг 3, получаем:

$A_7 = GOTO(A_2, *) = GOTO(\{E \rightarrow E \bullet * P\}, *) = CLOSURE (\{T \rightarrow T * \bullet P) = \{T \rightarrow T * \bullet P, P \rightarrow \bullet i, P \rightarrow \bullet (E)\},$

Включаем A_7 в φ.

$A_8 = GOTO(A_4, E) = GOTO(\{P \rightarrow (\bullet E), E \rightarrow \bullet E + T\}, E) =$

$= CLOSURE (\{P \rightarrow (E \bullet), E \rightarrow E \bullet + T\}) = \{P \rightarrow (E \bullet), E \rightarrow E \bullet + T\}.$

Включаем A_8 в φ.

Вычисляем множество $GOTO(A_4, T)$.

$A_8 = GOTO(A_4, T) = GOTO(\{E \rightarrow \bullet T, T \rightarrow \bullet T * P\}, T) =$

$= CLOSURE (\{E \rightarrow T \bullet, T \rightarrow T \bullet * P\}) = \{E' \rightarrow T \bullet, T \rightarrow T \bullet * P\}.$

Такое множество (множество A_2) уже есть в φ.

Множества $GOTO(A_4, P)$, $GOTO(A_4, ( )$ и $GOTO(A_4, i)$ также уже помещены в систему φ.

Множество $GOTO(A_5, X) = Ø$ для всех $X \in (N \cup \Sigma)$.

Продолжая аналогичным образом, включаем в систему φ множества:

$A_9 = GOTO(A_6, T) = GOTO(\{E \rightarrow E + \bullet T, T \rightarrow \bullet T * P\}, T) =$

$= CLOSURE (\{E \rightarrow E + T \bullet, T \rightarrow T \bullet * P\}) = \{E \rightarrow E + T \bullet, T \rightarrow T \bullet * P\},$

$A_{10} = GOTO(A_7, P) = GOTO(\{T \rightarrow T * \bullet P\}, P) =$

$= CLOSURE (\{ T \rightarrow T * P \bullet\}) = \{T \rightarrow T * P \bullet\},$

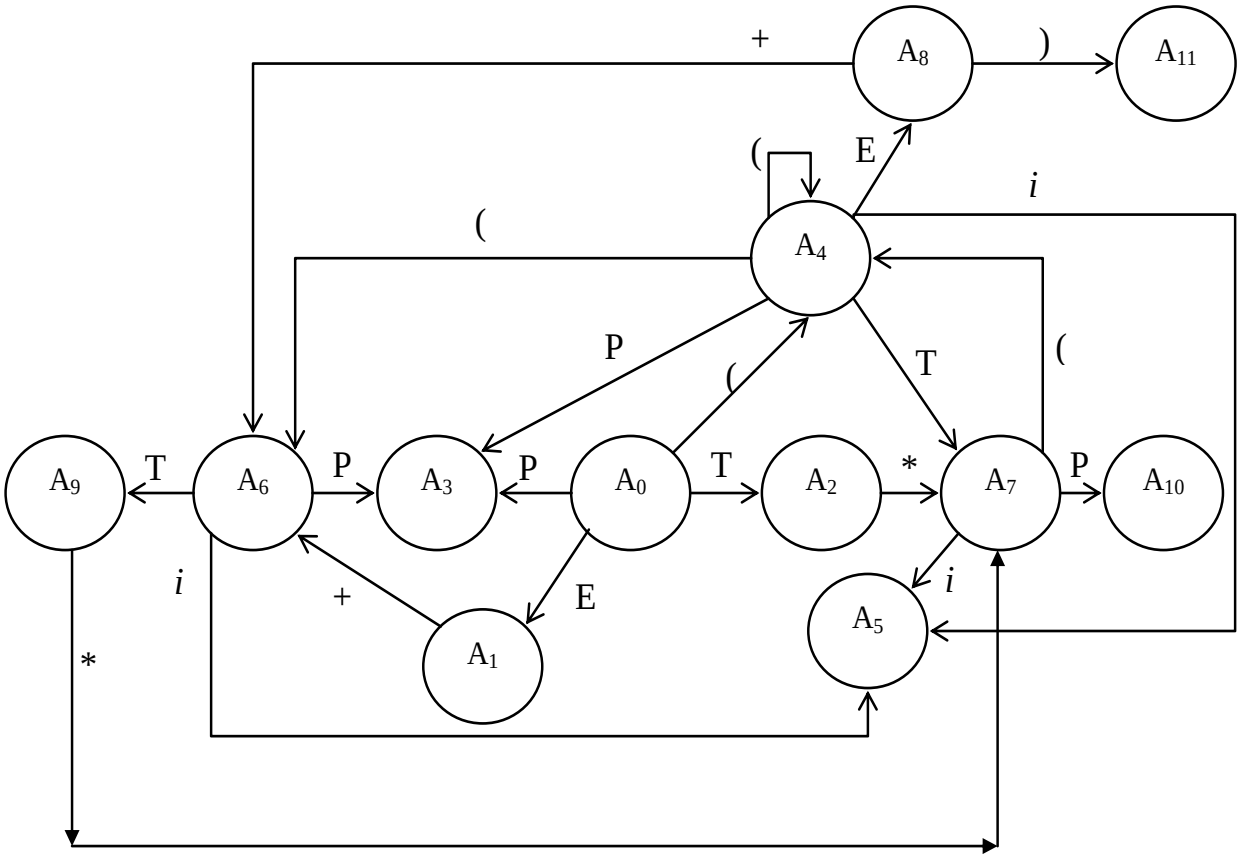
$A_{11} = GOTO(A_8, ( ) = GOTO(\{E \rightarrow (E \bullet)\}, ( ) =$

$= CLOSURE (\{ E \rightarrow (E) \bullet\}) = \{E \rightarrow (E) \bullet\}.$

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

A_0	$E' \rightarrow \bullet E$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * P$ $T \rightarrow \bullet P$ $P \rightarrow \bullet (E)$ $P \rightarrow \bullet i$
A_1	$E' \rightarrow E \bullet$ $E \rightarrow E \bullet + T$
A_2	$E \rightarrow T \bullet$ $T \rightarrow T \bullet * P$
A_3	$T \rightarrow P \bullet$
A_4	$P \rightarrow (\bullet E)$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * P$ $T \rightarrow \bullet P$ $P \rightarrow \bullet (E)$ $P \rightarrow \bullet i$
A_5	$P \rightarrow i \bullet$
A_6	$E \rightarrow E + \bullet T$ $T \rightarrow \bullet T * P$ $T \rightarrow \bullet P$ $P \rightarrow \bullet (E)$ $P \rightarrow \bullet i$
A_7	$T \rightarrow T * \bullet P$ $P \rightarrow \bullet (E)$ $P \rightarrow \bullet i$
A_8	$E \rightarrow E \bullet + T$ $P \rightarrow (E \bullet)$
A_9	$E \rightarrow E + T \bullet$ $T \rightarrow T \bullet * P$
A_{10}	$T \rightarrow T * P \bullet$
A_{11}	$P \rightarrow (E) \bullet$

Диаграмма переходов ДКА для активных префиксов грамматики G_0.



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

Практический интерес представляют LR(k)-грамматики, для которых k=0 или k=1, в случаях, когда k > 1 требуется построение большого числа вспомогательных множеств, поэтому применение LR(k)-анализатора не оправдано.

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.

Пример построения LR(k)-анализатора.

1. Построить управляющую таблицу M для заданной LR(k) грамматики G= (T, V, P, S), где P = {p_1,p_2,p_3,p_4,p_5,p_6}, p_1) S→ aAb, p_2) S→ c, p_3) A→ bS, p_4) A→ Bb, p_5) B→ aA, p_6) B→ c

2. Описать работу алгоритма.

1. Для построения управляющей таблицы M воспользуемся алгоритмом 1.4.12. ++++++++++++++++++++++++=

В табл. 3.12.1. приведен результат выполнения алгоритма.

Управляющая таблица M для этой грамматики приведена в табл. 1.3.

T	f(u) функция действия				g(X)					
	a	b	c	⊥	a	b	c	S	A	B
S_0				Д						
a_1	П	П	П		a_5	b_3	c_6		A_1	B_4
A_1	П	П	П			b_1				
b_1	C,1	C,1	C,1	C,1						
c_2	C,2	C,2	C,2	C,2						
b_3	П	П	П		a_1		c_2	S_3		
S_3	C,3	C,3	C,3	C,3						
B_4	П	П	П			b_4				
b_4	C,4	C,4	C,4	C,4						
a_5	П	П	П	П	a_5	b_3	c_6		A_5	B_4
A_5	C,5	C,5	C,5	C,5						
c_6	C,6	C,6	C,6	C,6						
⊥	П	П	П		a_1		c_2	S_0		

Рассмотрим, например, вторую строку таблицы M, отмеченную грамматическим вхождением a_1, которое представляет собой префикс правой части правила p_1. Пусть b – текущий входной символ. Согласно правилу p_1 за a_1 может следовать грамматическое вхождение нетерминала A_1 или символ, представляющий подцепочку, порождаемую вхождением A_1. Цепочка порождаемая вхождением A_1, может начинаться либо вхождением b_3 (правило p_3), либо вхождением B4 (правило p_4), которое в свою очередь порождает цепочку, начинающуюся с a_1 или c_6. Следовательно, для входного символа b в магазин можно втолкнуть только грамматическое вхождение b_3, которое является кодированным представлением префикса правой части правила p_3.

Строка таблицы, отмеченная маркером для магазина (⊥), соответствует начальной конфигурации алгоритма. В первый момент времени в магазин можно записать только грамматическое вхождение a_1 или c_2, 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). Функция действия f(u) не зависит от текущего входного символа, а определяется только верхним символом магазина, то есть для выбора нужного действия (переноса или свертки) алгоритм не должен заглядывать вперед на символы входной цепочки (k=0).

2. Работа алгоритма 3.11. 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.

Шаг 1. Начальная конфигурация алгоритма – ($\perp$, abcb$\perp$, ε) (магазина находится маркер для магазина, а текущим входным символом является символ a).

Для строки управляющей таблицы, отмеченной символом $\perp$, f(a)=ПЕРЕНОС и g(a) = a_1, поэтому:

- в магазин записывается символ a_i (грамматическое вхождение символа a в правую часть первого правила);
- входная головка сдвигается на один символ справа.

Алгоритм приходит в конфигурацию ($\perp a_1$, bcb$\perp$, ε).

Шаг 2. Для строки таблицы, отмеченной символом a_1, f(b)=ПЕРЕНОС и g(b) = b_3, следоательно, алгоритм перейдет в конфигурацию ($\perp a_1 b_3$, cb$\perp$, ε). Аналогично для магазинного символа b_3 и текущего символа входной цепочки магазин перейдет в конфигурацию ($\perp a_1 b_3 c_2$, b$\perp$, ε)

Шаг 3. Рассмотрим строку таблицы M, помеченную грамматическим вхождением c_2. В этом случае f(b)= (c, 2), значит, необходимо выполить свертку с использованием правила p_2) S→ c.

Правая часть этого правила содержит только один символ, поэтому:

- удаляем из магазина символ c_2;
- определяем значение функции переходов для символа S из левой части правила p_2 в строке таблицы M, отмеченной символом b_3, который стал верхним символом магазина. Значение функции переходов g(S) = S_3.

Алгоритм переходит в конфигурацию ($\perp a_1 b_3 S_3$, b$\perp$, 2) и в выходную цепочку записывается число 2 (номер использованного варианта).

Продолжая, получим следующую последовательность тактов:

($\perp a_1 b_3 S_3$, b$\perp$, 2) ⊢ ($\perp a_1 A_1$, b$\perp$, 23) ⊢ ($\perp a_1 A_1 b_1$, $\perp$, 23) ⊢ ($\perp a_1 S_0$, $\perp$, 231)

Заметим, что конфигурация ($\perp a_1 S_0$, $\perp$, 231) является заключительной, а цепочка 231 – правым разбором цепочки abcb.

Пусть анализируется цепочка aabc, содержащая синтаксическую ошибку, тогда последовательность тактов следующая:

($\perp$, aabc$\perp$, ε) ⊢ ($\perp$, aabc$\perp$, ε) ⊢ ($\perp a_1$,abc$\perp$, ε) ⊢ ($\perp a_1 a_5$, bc$\perp$, ε)
⊢ ($\perp a_1 a_5 b_3$, c$\perp$, ε) ⊢ ($\perp a_1 a_5 b_3 c_2$, $\perp$, ε) ⊢ ($\perp a_1 a_5 b_3 S_3$, $\perp$, 2) ⊢ ($\perp a_1 a_5 A_5$, $\perp$, 23) ⊢
($\perp a_1 B_4$, $\perp$, 235) ⊢ ОШИБКА.

f(ε) = ОШИБКА. Алгоритм должен выдать сообщение об ошибке и закончит работу.

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).

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 $\lambda = (f, g)$, 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 $g(\alpha,w) = i$, то правая часть i-го правила является суффиксом цепочки α.

Пусть имеется сентенциальная форма $\alpha X_1X_2\beta$, 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 $X_1 \bullet> X_2$. 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 $(\perp X_1 \dots X_m, a_1 \dots a_n, p_1 \dots p_r)$, где :

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

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

$p_1 \dots p_r$ – 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 $f(\alpha,aw)$ = 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 $\pi \in \{1,\dots,p\}^*$.

2. Если $f(\alpha\beta, w)$ = СВЕРТКА (С), $g(\alpha\beta, w) = i$ 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 $f(\alpha, w)$ = 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 •>, принимается решение о свертке, в противном случае делается перенос.

Таким образом, с помощью отношения •> 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
⊥bAb				1
αAad		4		
αcB	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:

$S \Rightarrow (p_1)\ bAb \Rightarrow (p_2)\ bcBb \Rightarrow (p_4)\ bcAdb \Rightarrow (p_3)\ bcaadb$

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},

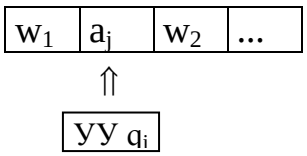
$\delta = Q \times \Sigma$ в $Q \times (\Sigma \cup \{$**B**,L,R$\})$ $q_i a_j \rightarrow q_k$ {**B**,L,R}

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.

4.3. Соотношения между грамматиками и языками

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 $A \rightarrow \varepsilon$, 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 $P_1 = \{S \rightarrow 0A1,\ 0A \rightarrow 00A1, A \rightarrow \varepsilon\}$ и $P_2 = \{S \rightarrow 0S1 \mid 01\}$

описывают один и тот же язык $L = L(G_1) = L(G_2) = \{ 0^n1^n \mid n>0\}$. 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. Формальные методы описания перевода

Глава 6. Разрработка 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

АТ-грамматика называется ***L-атрибутной*** тогда и только тогда, когда выполняются следующие условия:

- 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.
- Аргументами правила вычисления значения синтезированного атрибута символа из левой части правила вывода являются унаследованные атрибуты этого символа или произвольные атрибуты символов из правой части.
- 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^4.

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 $A \rightarrow BC$ 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.***

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 детерминированный анализатор для *LL*(1)-грамматик требует, чтобы *L*-атрибутная транслирующая грамматика, описывающая перевод, имела форму простого присваивания.

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 $a \leftarrow b$ или $x, y \leftarrow 1$.

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.***

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.

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 G^4.

Рассмотрим процедуру преобразования произвольной L-атрибутной транслирующей грамматики в эквивалентную L-атрибутную грамматику в форме простого присваивания. Смысл используемого здесь понятия эквивалентности объясняется далее:

1. Для каждой функции $f(x_1, x_2, \dots, x_n)$, входящей в правило вычисления атрибутов, связанное с некоторым правилом вывода грамматики, создать соответствующий ей операционный символ $\{f\}$, который определяется следующим образом:

   $\{f\}\ x_1\ x_2, \dots, x_n, p$ унаследованные $x_1, x_2, \dots, x_n$
   синтезированный p

   $p \leftarrow f(x_1, x_2, \dots, x_n)$

2. Для каждого некопирующего правила $Zi, Z2, \dots, Z_m \leftarrow f(y_1, y_2, \dots, y_n)$ связанного с некоторым правилом вывода грамматики, найти символы $a_1 \dots, a,,,$ которые не содержатся в этом правиле вывода, и вставить в его правую часть символ $\{f\}a_{1?}\ a_2, \dots, a_{n,r.}$ Заменить некопирующее правило на следующие $(n + 1)$ копирующих правил:

   $a_i \leftarrow y_i$ для каждого аргумента y_i,

   $z_1, z_2, \dots, z_m \leftarrow r$

   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 $y_1, y2, \dots, y_{n;}$
   - операционный символ должен располагаться *левее* всех символов правой части правила вывода, атрибутами которых служат $z_1, z_2, \dots, z_m$;
   - 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.

3. 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

(1) $S \rightarrow E_p$ {ОТВЕТ}$_r$

$r \leftarrow p$

(2) $E_p \rightarrow + E_q E_r$

$p \leftarrow q + r$

(3) $E_p \rightarrow *E_q E_r$

$p \leftarrow q^* r$

(4) $E_p \rightarrow c_r$

$p \leftarrow r$

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 атрибута ***A*** и ***B*** и один синтезированный атрибут ***R.*** Для операционного символа {СЛОЖИТЬ}$_{A, B, R}$ атрибутное правило имеет вид: $R \leftarrow A + B$, а для операционного символа {УМНОЖИТЬ}$_{A, B, R}$ — $R \leftarrow A \; X \; B$.

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:

E_p синтезированный p

{ОТВЕТ}$_r$, унаследованный r

{СЛОЖИТЬ}$_{A, B, R}$ унаследованный A, B

$R \leftarrow A + B$ синтезированный R

{УМНОЖИТЬ}$_{A, B, R}$ унаследованный A, B

$R \leftarrow A^* B$ синтезированный R

(1) $S \rightarrow E_p$ {ОТВЕТ}$_r$

$r \leftarrow p$

(2) $E_p \rightarrow + E_q E_r$ {СЛОЖИТЬ}$_{A, B, R}$

$A \leftarrow q$

$B \leftarrow r$

$p \leftarrow R$

(3) $E_p \rightarrow * E_q Er$ {УМНОЖИТЬ}$_{A, B, R}$

$A \leftarrow q$

$B \leftarrow r$

$p \leftarrow R$

(4) $E_p \rightarrow c_r$

$p \leftarrow r$

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 $(V_p \cup \{\perp\}) \times (\sum_I \cup \{\epsilon\})$ в множество, состоящее из следующих элементов:

1. (β, y), где $\beta \in Vp^*$ — цепочка из правой части правила транслирующей грамматики $A \rightarrow y\beta$, а $y \in \sum_a^*$;
2. *ВЫДАЧА (X),* где $X \in \sum_a$
3. ВЫБРОС;
4. ДОПУСК;
5. 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. $(x, X\alpha, \pi) \vdash (x, \beta\alpha, \pi(p))$, если $M(X, a) = (\beta, y)$. Верхний символ магазина X заменяется цепочкой $\beta \in Vp^*$, и в выходную цепочку дописывается цепочка операционных символов y. Входная головка при этом не сдвигается.

2. **(x, *X*α, π)⊢(x, α, π*X*)**, 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 G^T.

Описание алгоритма:

▪ Управляющая таблица ***M*** определяется на множестве (N ∪ $\sum_I$ ∪ $\sum_a$ ∪{⊥}) x ($\sum_I$ ∪ {ε}) по следующим правилам:

1. Если ***A→yβ*** - правило вывода грамматики G^T, где ***y*** ∈ $\sum_a^*$, а ***y*** — либо **ε**, либо цепочка, начинающаяся с терминала или нетерминала, то ***M{A, a)*** = **(β,** ***y)*** для всех ***a*** ≠ **ε,** принадлежащих множеству FIRST(**β**).

   Если **ε** ∈ FIRST(**β**), то ***M(A, b)*** = **(β,** ***y) для*** всех ***b*** ∈ FOLLOW (A).

   Заметим, что при вычислении FIRST(**β**) операционные символы, входящие в цепочку **β**, вычеркиваются.
2. ***M(X, a)* = *ВЫДАЧА(X)*** для всех **x** ∈ $\sum_a$ и ***a*** ∈ $\sum_i$ ∪ **{ε}.**
3. *M(a, a)* = ***ВЫБРОС для*** всех ***a*** ∈ $\sum_i$.
4. **M (⊥ , ε)** = **ДОПУСК**
5. В остальных случаях ***M(X, a)* = *ОШИБКА*** для ***X*** ∈ **(N ∪**$\sum_i$ ∪ $\sum_a$ ∪ **{⊥}) и *a*** ∈ $\sum_i$ ∪**{ ε }.**

Построим управляющую таблицу для транслирующей грамматики, описывающей перевод инфиксных арифметических выражений в ПОЛИЗ. Эта транслирующая грамматика получена из входной LL(1)-грамматики G_1 путем включения в нее операционных символов {i}, {+} и {*}:

(1)	***E → TE′***	***(5)***	***T′ → * P{*} T′***
(2)	***E′ → + T{+} E′***	***(6)***	***T′ → ε***
(3)	***E′ → ε***	***(7)***	***P → i {i}***
(4)	***T → PT′***	***(8)***	***P -> (E)***

Управляющая таблица должна содержать 14 строк, помеченных символами из множества **N ∪**$\sum_i$ ∪ $\sum_a$ ∪ **{⊥}**, и 6 столбцов, помеченных символами из множества $\sum_i$ ∪**{ ε }.** Построение управляющей таблицы для строк, отмеченных символами из множества **N ∪**$\sum_i$ ∪ **{⊥}**, 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**	**+ T{+} E′**		**ε, ε**
T	**PT′, ε**	**PT′, ε**				
T′			**ε, ε**	**ε, ε**	***P{*}T′, ε**	**ε, ε**
P	**i{i}, ε**	**(E), ε**				
1	*ВЫБРОС*					
(		*ВЫБРОС*				
)			*ВЫБРОС*			
+				*ВЫБРОС*		
*****					*ВЫБРОС*	
⊥						*ДОПУСК*
{i}	*ВЫДАЧА ({i})*					
{+	*ВЫДАЧА***({+})**					
{*	*ВЫДАЧА({*})*					

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 ***n*** 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 нетерминал.

Таблица 1.2. Значения источников копирующих правил

Источник	**Доступ к источнику**
Константа	Значение всегда доступно
Унаследованный атрибут нетерминала из левой части правила	Значение источника находится в соответствующем поле верхнего символа магазина
Синтезированный атрибут операционного символа, который не вталкивается в магазин	Значение источника вычисляется в соответствии с атрибутным правилом
Синтезированный атрибут вталкиваемых в магазин нетерминала и операционного символа	Недоступен
Атрибут входного символа	Недоступен

Приемниками атрибутных правил L-атрибутной транслирующей грамматики могут быть только синтезированные атрибуты нетерминалов из левой части правил вывода и унаследо-

ванные 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 $M(X, a) = (\beta, y)$.

Таблица 1.3. Значения приемников атрибутных правил

Приемник	Поле
Унаследованный атрибут нетерминала и операционного символа, вталкиваемых в магазин	Соответствующее поле вталкиваемого символа
Синтезированный атрибут нетерминала из левой части правила	Все поля в списке, на который указывает соответствующее поле магазинного символа

При выполнении перехода L-атрибутный ДМП-процессор выполняет следующие атрибутные действия:

1. 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.
2. Если источник копирующего правила доступен, то значение источника помещается в соответствующее поле магазинного символа.
3. 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.

Рисунок 1.4. Разработка и реализация синтактически управляемого перевода

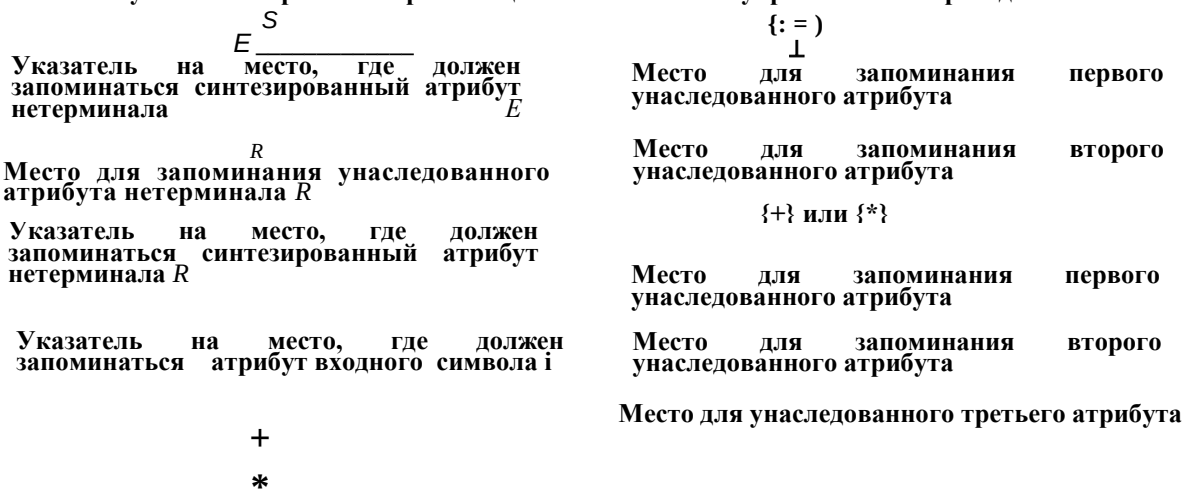


Таблица 1.5. Управляющая таблица для транслирующей грамматики, полученной из исходной грамматики

S	***i*:=*E*{:=}, ε**				
E	***iR*, ε**				
R			**+ i{+}** *R*, **ε**	***i{*}** *R*, **ε**	**ε, ε**
/	*ВЫБРОС*				
:=		*ВЫБРОС*			
+			*ВЫБРОС*		
*				*ВЫБРОС*	
⊥					*ДОПУСК*
{:=}	*ВЫДАЧА*($\{:=\}_{P,q}$)				
{+}	*ВЫДАЧА*($\{+\}_{P,}$ **q. r**)				
{*}	*ВЫДАЧА*($\{*\}_{P,}$ **q. r**)				

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 G_1 и имеет следующие правила:

***S* → *E*⊥**
***E* → *TE*'**
***E*' → + T { + } *E* ' | ε**
***T* → *PT* '**
***T*' → * *P* { * } *T* ' | ε**
***P* → *(E)* | *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).**

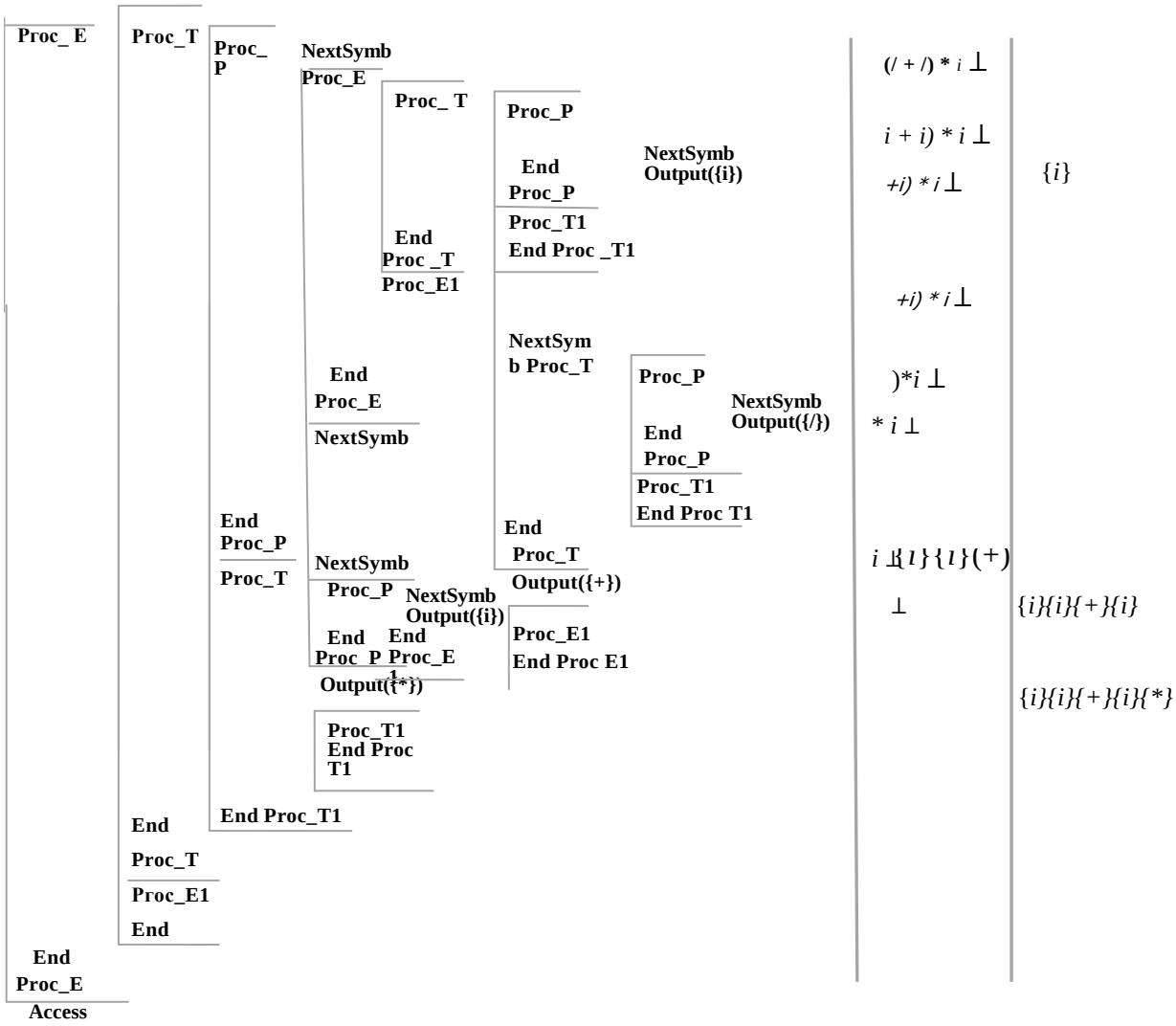


Рисунок 1.2. Порядок вызова процедур при переводе цепочки (i + i) * i ⊥

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:

$R_{p1,\,t2} \rightarrow + i_{q1} \{*\}_{p2,\,q2,\,r1} R_{r2,\,r1}$

$R_{p1,\,t2} \rightarrow + i_{q1} \{*\}_{p2,\,q2,\,r1} R_{r2,\,r1}$

$R_{p1,\,p2} \rightarrow \varepsilon$

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.

1. $A \rightarrow B_x C_y D_z$

$y, z \leftarrow x$

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 ($A \rightarrow B_x\ C_x D_z$), которое не требует правила вычисления атрибута ***x.***

2. $A_x \rightarrow B_y \{f\}_z\ Y$

$y, z \leftarrow x$

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.

3. $A_{x,\,y} \rightarrow a, B_z\ C_t$

$y, z, t \leftarrow x$

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:

$A_{x,y} \rightarrow a\ B_y\ C_y$

$y \leftarrow x$

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;**

- ***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;

- ***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:

E_t синтезированный t

$R_{p,t}$ унаследованный p синтезированный t

Атрибуты операционных символов унаследованные.

(0) $S0 \rightarrow S \perp$

(1) $S \rightarrow I_a := E_b \{:=\}_{a,b}$

(2) $E_c \rightarrow I_d R_{d,c}$

(3) $R_{p,t} \rightarrow i_{q1} \{+\}_{p,q1,r1} R_{r1,t}$
$r_1 \leftarrow$ GETNEW

(4) $R_{p,t} \rightarrow x\, i_{q2} \{*\}_{p,q2,r2} R_{r2,t}$
$r_2 \leftarrow$ GETNEW

(5) $R_{p,t} \rightarrow \varepsilon$
$t \leftarrow p$

С учетом выполненных преобразований процедура на языке Pascal, реализующая 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;
```

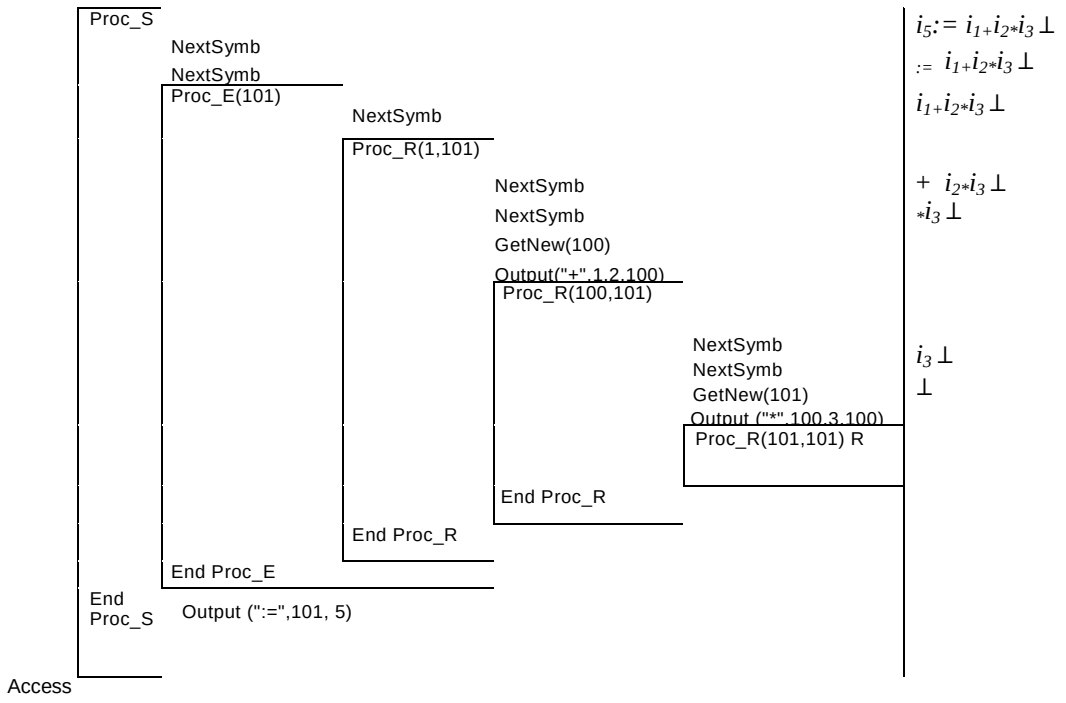
```
                NextSymb;
                GetNew(r2);
                Output({+}, p, q2, r2);
                Proc_R (r2, t)
            end
        else
            Error;
        end
        else
        t  := p
        end; {Proc_R}
        begin
        if ClassSymb = '*'
        then begin
```

{ 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.

Рисунок 1.3. Порядок вызова процедур при переводе цепочки $i_5 := i_1 i_2 * i_3 \perp$



4. S-атрибутный ДМП-процессор

4.1. Математическая модель восходящего ДМП-процессора

Для любого восходящего синтаксического анализатора, рассматриваемого в данном учебнике последовательность операций переноса и свертки, выполняемых при обработке допустимых входных цепочек можно описать с помощью 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 $G_Q = (\{E, T\backslash P\}, \{i, +, *, (,)\}, P, E)$, где $P = \{E \to E + T, E \to T, T \to T^* P, P, P \to i, P \to (E)\}$, будет выглядеть следующим образом:

(1) **E → E + T{СВЕРТКА,** 1}
(2) **E →T{СВЕРТКА,** 2**}**
(3) **T→ T* P {СВЕРТКА,** 3**}**
(4) **T→ P {СВЕРТКА,** 4}
(5) ***P*** **→** *i* ***{СВЕРТКА,*** 5}
(6) **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 ЕР**ТКА 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.

4.2 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 дл представления атрибутов. Так же, как и в ***L***-атрибутном ДМП-процессоре, примем, что магазинный символ с атрибутами представляется в магазине ***(n*** + 1)-ой ячейками, верхняя из которых содержит имя символа, остальные — поля для атрибутов. Поля магазинного символа, предназначенные для атрибутов, ***заполняютс*** значениями атрибутов в момент ***вталкивания символа в магазин*** и не изменяются до момента выталкивания его и магазина.

В S-атрибутном ДМП-процессоре ***операция переноса*** расширяется таким образом, что значения атрибуто переносимого входного символа помещаются в соответствующие поля вталкиваемого при перенос магазинного символа.

При выполнении ***операции свертки*** для правила с номером ***i*** верхние символы магазина представляют собо правую часть *i*-го правила вывода входной грамматики, а поля магазинных символов содержат значени атрибутов соответствующих символов грамматики.

Расширенная операция свертки использует эти значения для вычисления значений всех атрибуто операционных символов, связанных с правилом вывода транслирующей грамматики, и значений всех атрибуто нетерминала из левой части правила. ***Значения атрибутов операционных символов*** используются для выдач результатов в выходную ленту или выполнения других действий, определяемых этими символами. ***Атрибут нетерминала из левой части правила вывода*** записываются в соответствующие поля магазинного символа, которы соответствует этому нетерминалу и вталкивается в магазин во время свертки.

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..

A).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());
```

В). Определить последовательность тактов МП-автомата для выделенной цепочк.

2.2. А). Построить расширенный МП-автомат, используя алгоритм 3.9.

В). Определить последовательность тактов расширенного МП-автомата при анализе входной выделенной цепочки Р.

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-ориентированную 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. $L=\{10\omega_1 0\text{-}1\omega_2 \mid \omega_1 \in \{0,1\}^+, \omega_2 \in \{1,2\}^*\}$
22. $L=\{1\text{-}1\omega_1 01\omega_2 \mid \omega_1 \in \{1,2\}^+, \omega_2 \in \{1,2\}^*\}$

Лабораторные работы 3-6

A).Устранить из грамматики G =(T, V, P, S) бесполезные символы, где

1. P={S → b, S → F, S→ cFB, A→ Ab, A→ c, B → cB, F→ Ca, C→ d}
2. P={S → b, S→ cAB, A→ Ab, A→ c, B → cB, C→ Ca, F→ 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. *Р.Хантер. Проектирование и конструирование компиляторов.* Пер. с англ. – М.: Финансы и статистика, 1984.
3. *Н.Вирт.* Алгоритмы + структуры данных = программы. Пер. с англ. – М.: МИР, 1985.
4. *C.Moore.* Dynamical Recognizers: Real-time Language Recognition by Analog Computers. Theoretical Computer Science **201**, 1998, pp. 99-136.
5. *W.Tabor.* Fractal Encoding of Context Free Grammars in Connectionist Networks. University of Connecticut Expert Systems 17(1), 2000, pp. 41-56
6. *А.С.Семенов.* Построение класса фрактальных систем по шаблону на примере дерева Фибоначчи // Изв.РАН. Информационные технологии и Вычислительные системы. – М.: N2, 2005 стр.10-17.

7. *А.С.Семенов* Класс фрактальных автоматов для распознавания КС-языков // Программирование (рукопись).